MASTER IN COMPUTER SCIENCE

Strictness analysis for a class of second order functional languages

Burlet, Frédéric

Award date:
2002

FUNDP
Institut d'Informatique

Rue Grandgagnage, 21
B - 5000 NAMUR (Belgique)

Strictness Analysis For A Class Of Second Order Functional Languages

Frédéric BURLET

Under the management of Baudouin Le Charlier

Institut d'Informatique
Facultés Universitaires Notre-Dame de la Paix
Namur

Juin 2002

Abstract

Strictness analysis is a method of static analysis based on abstract interpretation for functional programs. The purpose of this technique is to transform call-by-name into call-by-value when the result of function call are identical. So, the semantics of call-by-name is unchanged and the evaluation of the result is more efficient. This analysis is applied to a second order functional language.

Résumé

L'analyse de strictness est une méthode d'analyse statique de programme basée sur l'interpretation abstraite. Le but de cette technique est de transformer le passage par nom des arguments dans les appels de fonction par le passage par valeur quand les résultats des appels sont égaux. Ainsi, la sémantique du passage par nom ne s'en trouve pas modifiée et les calculs des résultats se fait de manière plus efficace. Cette analyse est appliquée à un langage fonctionnel du second ordre.

I would like to first thank my promoter, Baudouin Le Charlier, without whom this work would not exist; Mads Rosendahl, my training course supervisor, with whom I have hugely learned during my stay in Denmark; Steve Uhlig for his numerous advices and corrections in English. I also would like to thank Vincent Letocart for his marvellous advices given through IRC during my training course and all the persons that have indirectly supported me morally during this hardship, namely my parents, my friends and many other peoples that I have perhaps forgotten.

Contents

Introduction

The main subject of this work is functional languages. A functional language is a language where all commands are considered as expressions and a program as a series of function declarations. We distinguish different types of functions: *first order functions* which are functions that take solely variables as arguments and *higher order functions* which are functions that take not only variables as arguments but functions too. Higher order is also divided into several orders: 2^{nd}, 3^{rd} and so on ... A second order functional language is a language where functions may take solely first order functions as arguments (and obviously variables). A third order functional language is a language where functions may take either first order functions or second order functions as arguments, and so on for the n^{th} order.

We may categorize functional languages in two families: *strict* languages such as, for example, `caml` (`http://caml.inria.fr`) and `Lisp` and *non-strict* or also called *lazy* languages such as `Haskell` (`http://www.haskell.org`) and `Miranda`. Both differ by their parameter passing. The former implements *call-by-value* while the latter implements *call-by-name*. Given the function f below picked from [Myc80] [Pol96], let us examine how both work:

$$\underline{\text{let}}\ f(x,y) = \underline{\text{if}}\ x = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ f(x-1, f(x,y))$$

The principle of call-by-value is to evaluate arguments of function call before performing the call. We have the following developments:

$$\begin{aligned}
f(0,y) &= 1 \\
\\
f(1,y) &= \underline{\text{if}}\ 1 = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ f(0, f(2,y)) \\
&= f(0, f(2,y)) \\
&= f(0, \underline{\text{if}}\ 1 = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ f(0, f(2,y))) \\
&= f(0, f(0, f(2,y))) \\
&= f(0, f(0, \underline{\text{if}}\ 1 = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ f(0, f(2,y)))) \\
&= f(0, f(0, f(0, f(2,y)))) \\
&= \text{and so on } \ldots
\end{aligned}$$

If we summarize the function results we see that the function gives a result

only when $x = 0$:

$$f(x,y) = \begin{cases} 1 & \text{if x} = 0 \ , \\ \bot & \text{otherwise.} \end{cases}$$

On the other hand, the principle of the call-by-name is to evaluate arguments of function call as late as possible. If we make the developments, we have the following:

$$\begin{aligned} f(0,y) &= 1 \\ &= \\ f(1,y) &= \underline{\text{if}}\ 1 = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ f(0,f(2,y)) \\ &= f(0,f(2,y)) \\ &= \underline{\text{if}}\ 0 = 0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ f(0,f(2,y)) \\ &= 1 \end{aligned}$$

If we summarize the function results we have that for all x and y that $f(x,y) = 1$.

By definition, call-by-value is fast, efficient and easy to implement. We may also imagine that arguments of function calls are evaluated in parallel such that the computation of function calls is improved. But as seen in the previous example, call-by-value may not terminate. It is not the case in call-by-name. Its semantics is a natural way to evaluate expressions in a functional language. The problem of call-by-name is that it is space- and time-consuming because arguments are lugged around during all the evaluation of the function call. An optimization for a lazy functional language should be to transform its costly evaluation mechanisms into a call-by-value.

Strictness analysis is a method of static analysis based on abstract interpretation for functional languages. Its goal is to replace call-by-name by call-by-value when the results of function calls are identical. It studies strictness properties of a functional language. We will say that a function is strict on one of its arguments if when the evaluation of this argument is undefined then the result of the function call is undefined. Strictness analysis detects strict functions for which tranformation may be applied.

This work is structured as follows:

- *Chapter 1* introduces all the mathematical framework that we use later in this work. The three most important notions are *domain*, *fix-point* and *strict functions*.

- *Chapter 2* introduces a first order functional language. We present different implementations of the same language and show the differences between them. We also study a specific fix-point algorithm which is the minimal function graph algorithm [DJM86] [DJR97]. Then we construct a strictness interpreter using minimal function graph.

- *Chapter 3* dresses the development of a higher order functional language. We have limited this language to the second order. We do the same development than in the previous chapter but we specify it more formally and explain difficulties encountered during implementation.
- Finally, we conclude our work and present further work.

Chapter 1

Mathematical Framework

1.1 Introduction

This chapter is devoted to the description of the mathematical concepts used in the remainder of this document. We introduce some basic concepts that can be found in the paper [Ros01]: domains, monotonous functions, continuous functions and fix-points. These concepts allow us to introduce a fundamental theorem: *the fix-point theorem.* This is an important tool in mathematical semantics and in abstract interpretation.

1.2 Complete Partial Order

Complete partial orders (CPO) or *domains* are playing an important role in the study of strictness analysis and fix-point definition. The aim of this section is to explain this notion. Before getting to it, we first introduce some useful definitions.

Definition 1.1 (Partial order relation)
A relation $\leq$ on a set E is a partial order relation if and only if this relation respects the following properties:
(Reflexivity) $\forall x \in E : x \leq x$
(Transitivity) $\forall x, y, z \in E : x \leq y \ \wedge \ y \leq z \ \Rightarrow \ x \leq z$
(Antisymmetry) $\forall x, y \in E : x \leq y \ \wedge \ y \leq x \ \Rightarrow \ x = y$

In this case, we call the couple $(E, \leq)$ a **partial order**.

Definition 1.2 (Lower bound)
Given a partial order $(E, \leq)$ and $S \subseteq E$, m is a lower bound of S if and only if $\forall x \in S \quad m \leq x$

Definition 1.3 (Greatest lower bound)
Given a partial order $(E, \leq)$, m is the greatest lower bound or infimum of $S \subseteq E$ if:

- *m is a lower bound of S*
- *$\forall x$ lower bound of S, we have $x \leq m$*

We will use the notation $\sqcap S$ to indicate a greatest lower bound of S.

In the same way, we can define similarly the upper bound and the least upper bound of a set. Remark here that if the greatest lower bound (of least upper bound) exists, it is unique.

Definition 1.4 (Chain)
Given a partial order $(E, \leq)$, we call $(x_i)_{i \in I}$ a chain of E if and only if $\forall i \in I, x_i \in E$ and $x_i \leq x_{i+1}$

We will often use this notion, thus we introduce a notation for it [Ros01]. We define $\mathsf{chains}(E)$, the set of chains from E and write $(x_i) \in \mathsf{chains}(E)$ for a chain $x_1 \leq x_2 \leq \ldots$ in E. A chain may be considered as an increasing sequence of elements in a domain E.

Now, we define now the notion of **complete partial order**, more usually called **domain**.

Definition 1.5 (Domain)
A pair $(E, \leq)$ is called a domain if and only if
- *E is a non-empty set;*
- *$\leq$ is a partial relation order on E;*
- *E has a lower bound called $\bot_E$;*
- *All chains $x_1 \leq x_2 \leq \ldots \leq x_i \leq \ldots$ in E have a least upper bound $\sqcup_i x_i$ in E.*

Note that every set E can be extended to a domain by adding a least element $\bot$ and using a special ordering called *flat ordering*, noted $\sqsubseteq$ and defined by:

$$\forall x, y \in E \cup \{\bot\} : x \sqsubseteq y \iff (x = \bot) \vee (x = y)$$

In this case, we say that the set E is lifted with the bottom value ($\bot$).
To denote the minimal element of a domain, we use the symbol $\bot$.

1.3 Continuity

As seen later in this chapter, continuity is an important notion to ensure the existence of a fix-point.

Definition 1.6 (Monotonous function)
Given $(E, \leq_E)$ and $(F, \leq_F)$ two domains, $f : E \to F$ is monotonous if and only if $\forall e_1, e_2 \in E : e_1 \leq e_2 \Rightarrow f(e_1) \leq f(e_2)$.

Definition 1.7 (Continuous function)
Given $(E, \leq_E)$ and $(F, \leq_F)$ two domains, $f : E \to F$ is continuous if and only if f is monotonous and if $\forall (e_i) \in chains(E)$ f verifies the equality:

$$\bigsqcup_i^f (f(e_i)) = f(\bigsqcup_i^e (e_i))$$

Note that the composition of monotonous (*resp.* continuous) function is a monotonous (*resp.* continuous) function.

1.4 Fix-point

Now we first introduce a definition of fix-point. Then, we present a proposition which binds the notions of fix-point and domain.

Definition 1.8 (Fix-point)
Given $f : E \to E$ and e an element of E. The point e is a fix-point of the function f if and only if $f(e) = e$. If for all points x in E such that $f(x) = x$ we have $e \leq x$, e is the least fix-point.

Theorem 1.1 (Fix-point theorem)
A continuous function f on a domain $(E, \leq)$, $f : (E, \leq) \to (E, \leq)$ has a least fix-point which we can find as

$$\bigsqcup_i f^i(\bot)$$

PROOF.
The proof has three parts:

- We have a well-defined chain: $\bot \leq f(\bot) \leq f^2(\bot) \leq \ldots$
- We have a fix-point: $f(\bigsqcup_i f^i(\bot)) = \bigsqcup_i (f^i(\bot))$
- The fix-point is the least one: $\forall x \in E\, f(x) = x \Rightarrow \bigsqcup_i f^i(\bot) \leq x$

$\Diamond$

For a detailed proof, we refer to [Ros01].

The theorem says that for any domain E we have a function fix : $(E \to E) \to E$ which finds the least fix-point of a function.

1.5 Domain constructions

We will see in this section some domain constructions and their properties.

1.5.1 Cartesian product

Definition 1.9 (Cartesian product)
Given n domains $(E_1, \leq_{E_1}), \ldots, (E_n, \leq_{E_n})$, we define the Cartesian product $(E_1 \times \ldots \times E_n, \leq_{E_1 \times \ldots \times E_n})$ as the domain of tuples of values from set E_1 to E_n where :
- $E_1 \times \ldots \times E_n = \{\langle e_1, \ldots, e_n \rangle \mid e_1 \in E_1 \wedge \ldots \wedge e_n \in E_n\}$
- $\langle e_1, \ldots, e_n \rangle \leq_{E_1 \times \ldots \times E_n} \langle e'_1, \ldots, e'_n \rangle \iff e_1 \leq_{E_1} e'_1 \wedge \ldots \wedge e_n \leq_{E_n} e'_n$

This definition allows us to introduce some notation with respect to Cartesian product. If $\forall i \in \{1, \ldots, n\} : E_i = E$ then we note the Cartesian product $E_1 \times \ldots \times E_n$ as E^n and $\forall \rho, \eta \in E : \rho \leq \eta \iff \forall i \quad \rho_i \leq \eta_i$ where subscripted notation is used to select the i^{th} component of the tuple.

We may state some outstanding properties of Cartesian product:

Theorem 1.2 (Cartesian product of domain)
The Cartesian product of domains is a domain.

The proof of this theorem may be done by demonstrating these following points:

- The relation $\leq_{E_1 \times \cdots \times E_n}$ is a partial order;
- There is a least element in $E_1 \times \cdots \times E_n$;
- All chains of $E_1 \times \cdots \times E_n$ has a least upper bound in $E_1 \times \cdots \times E_n$.

This first property show that Cartesian product preserves the domain definition.

The next property shows that we may construct a continuous function on a Cartesian product from several continuous functions on domains.

Theorem 1.3 (Continuous function on Cartesian product)
Given $(E_1, \leq_{E_1}), \ldots, (E_n, \leq_{E_n})$ and $(F_1, \leq_{F_1}), \ldots, (F_n, \leq_{F_n})$ domains. If functions $f_1 : E_1 \to F_1, \ldots, f_n : E_n \to F_n$ are continuous then the function f defined by

$$\begin{array}{rccc} f: & E_1 \times \ldots \times E_n & \to & F_1 \times \ldots \times F_n \\ & \langle e_1, \ldots, e_n \rangle & \rightsquigarrow & \langle f_1 e_1, \ldots f_n e_n \rangle \end{array}$$

is continuous for Cartesian product.

This second property shows that Cartesian product preserves the continuity definition.

1.5.2 Sum of domains

Definition 1.10 (Sum of domains)
Given n domains $(E_1, \leq_{E_1}), \ldots, (E_n, \leq_{E_n})$, we can construct the sum (or disjoint union) $(E_1 + \cdots + E_n, \leq_{E_1+\cdots+E_n})$ as
- $E_1 + \cdots + E_n = \{\perp_{E_1+\cdots+E_n}\} \cup \{\langle i, e_i\rangle | i \in \{1, \ldots, n\} \wedge e_i \in E_i\}$
- $\forall x \in E_1 + \cdots + E_n \quad \perp_{E_1+\cdots+E_n} \leq_{E_1+\cdots+E_n} x$
- $\langle i, e_i\rangle \leq_{E_1+\cdots+E_n} \langle i, e_i'\rangle \iff e_i \leq_{E_i} e_i' \quad \forall i \in \{1, \ldots, n\}$

Theorem 1.4 (Sum domain)
The sum of domains is a domain.

The steps to prove this theorem are the same than in the Cartesian product case:

- The relation $\leq_{E_1+\cdots+E_n}$ is a partial order;
- There is a least element in $E_1 + \cdots + E_n$;
- All chains of $E_1 + \cdots + E_n$ has a least upper bound in $E_1 + \cdots + E_n$.

1.5.3 Function domain

Theorem 1.5 (Function domain)
Given a non-empty set E and a domain $(F, \leq_F)$ we can construct the function domain $(E \to F, \leq_{E\to F})$ of continuous functions from E to F with ordering:

$$f \leq_{E\to F} g \iff \forall x \in E,\ f(x) \leq_F g(x)$$

1.6 Strict functions

We define a strict function as follows:

Definition 1.11 (Strict function)
Given $(E_1, \leq), \ldots, (E_n, \leq)$ and $(E, \leq)$ (n+1) domains and given $f : E_1 \times E_2 \times \ldots \times E_n \to E$ a continuous function. We will say that the function f is strict in its i-th argument if and only if $\forall (e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n) \in E_1 \times E_{i-1} \times E_{i+1} \times \ldots \times E_n$,
$f(e_1, \ldots, e_{i-1}, \perp_{E_i}, e_{i+1}, \ldots, e_n) = \perp_E$

In a lazy functional language with this property, if the computation of the argument does not terminate then the computation of the result of the function will not terminate either. Hence, if we compute the argument before the call (it is the case in the call-by-value) we then have two possibilities: either the argument can be evaluated and no harm has been done or the evaluation of the argument will fail to terminate. In the latter case, with a

strict function, we know that the computation of the result of the function would have failed anyway. The only difference is that it now may happen a bit earlier. Strictness of a function means that we may use a call-by-value strategy rather than call-by-name and this should hopefully make it possible to implement the function more efficiently.

An operational semantics based on call-by-value implies that all functions are strict in their arguments.

Chapter 2

First-order case

2.1 Introduction

In this chapter, we present the step-by-step development of a strictness interpreter of a first order functional language. The aim is to present basic concepts to show how the implementation is constructed for a semantics. First we describe the syntax of this language. Then we explain issues of the transition from a *call-by-value* interpreter to a *call-by-name* interpreter. We try to explain difficulties we have encountered when going from one to the other. For these first implementations we use simple types like integers or strings. The next step is to upgrade both implementations with the type 'list' and see what we have to change to their first implementation. We also see another approach of the fix-point algorithm. This one is called the *minimal function graph* approach and computes the value of a function call only if necessary. Finally, we present a strictness interpreter using minimal function graph.

2.2 A call-by-value interpreter

We describe in this section the semantics of a simple call-by-value interpreter of a first order functional language.

We work with a set of values $\mathcal{D}$. No matter the instantiation of the set but $\mathcal{D}$ must contain at the time of implementation single values. For example, we can imagine that $\mathcal{D}$ is the union between integers and strings:

$$\mathcal{D} = \mathbb{Z} \cup \mathbb{S}$$

We define the domain $\mathcal{D}_\perp$ which is the set $\mathcal{D}$ lifted with the bottom value: $\mathcal{D}_\perp = \mathcal{D} \cup \{\perp\}$.

In a simple functional language, we distinguish two types of elements: expressions and user-defined functions declaration.

An expression is either a constant (c_i) or a variable (x_i) or a primitive applied to several arguments ($p(expr_1, \ldots, expr_n)$) or a user-defined function call ($f(expr_1, \ldots, expr_n)$):

$$\begin{array}{rcl} expr & ::= & c_i \\ & | & x_i \\ & | & p_i(expr_1, \ldots, expr_n) \\ & | & f_i(expr_1, \ldots, expr_n) \end{array}$$

A program is composed by user-defined functions descriptions.

$$\begin{array}{rcl} prog & ::= & f_1(x_1, \ldots, x_n) = expr_1 \\ & | & f_2(x_1, \ldots, x_n) = expr_2 \\ & \vdots & \vdots \\ & | & f_p(x_1, \ldots, x_n) = expr_p \end{array}$$

We point out that all functions have the same number of arguments. Well, this is not an issue indeed. We can replace non-used arguments by dummy variables to solve the problem. This syntax is very short and easy to use for implementation.

To implement the interpreter, we need an environment to keep in memory to which value a variable name or a function name points. A variable takes its value from the set $\mathcal{D}$. Function arguments take also their value from $\mathcal{D}$ but a function call returns a value from the domain $\mathcal{D}_\perp$: if the result of a function call is $\perp$ this means that the function call does not terminate. We define a variable environment and a functional environment as follows:

$$\begin{array}{rcll} \rho & \in & \mathcal{D}^n & \text{variable environment} \\ \phi & \in & \Phi \equiv (\mathcal{D}^n \to \mathcal{D}_\perp)^p & \text{functional environment} \end{array}$$

We need some functions to evaluate expressions and the meaning of a program. In order to define an expression evaluation function, we need the variable environment to be able to get the value of a variable and a functional environment to be able to evaluate the value of a function call. The result of the evaluation of an expression may be $\perp$ because the evaluation of a function call may not terminate. We call $\mathcal{E}$ the evaluation function for expressions which takes a variable environment and a functional environment. Given an expression it returns a value of $\mathcal{D}_\perp$:

$$\mathcal{E}[\![expr]\!] : \Phi \to \mathcal{D}^n \to \mathcal{D}_\perp$$

Note before describing the semantics that a call-by-value interpreter evaluates the arguments of a function first, before performing the call. As described in section 1.6, an operational semantics based on call-by-value

implies that all functions are strict in their arguments. Therefore, a call-by-value interpreter evaluates expressions in this way:

$$
\begin{array}{l}
\mathcal{E}[\![c_i]\!]\ \phi\ \rho = d_i \\
\mathcal{E}[\![x_i]\!]\ \phi\ \rho = \rho_i \\
\mathcal{E}[\![p_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho = \\
\qquad \texttt{strict}\ [\![p_i]\!](\mathcal{E}[\![expr_1]\!]\ \phi\ \rho, \ldots, \mathcal{E}[\![expr_n]\!]\ \phi\ \rho) \\
\underline{\mathcal{E}[\![\texttt{if}}(expr_1, expr_2, expr_3)]\!]\ \phi\ \rho = \\
\qquad \texttt{cond}(\mathcal{E}[\![expr_1]\!]\ \phi\ \rho, \mathcal{E}[\![expr_2]\!]\ \phi\ \rho, \mathcal{E}[\![expr_3]\!]\ \phi\ \rho) \\
\mathcal{E}[\![f_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho = \\
\qquad \texttt{strict}\ \phi_i(\mathcal{E}[\![expr_1]\!]\ \phi\ \rho, \ldots, \mathcal{E}[\![expr_n]\!]\ \phi\ \rho)
\end{array}
$$

where

$$
\begin{array}{rrcl}
\texttt{strict}: & (\mathcal{D}^n \to \mathcal{D}_\perp) & \to & (\mathcal{D}_\perp^n \to \mathcal{D}_\perp) \\
& f & \rightsquigarrow & \texttt{if}\ (x_1 = \perp \vee x_2 = \perp \vee \ldots \vee x_n = \perp) \\
& & & \texttt{then}\ \perp\ \texttt{else}\ f(x_1, \ldots, x_n) \\
\texttt{cond}: & \mathcal{D}_\perp^3 & \to & \mathcal{D}_\perp \\
& (v_1, v_2, v_3) & \rightsquigarrow & \texttt{if}\ v_1\ \texttt{then}\ v_2 \\
& & & \texttt{else if not}\ v_1\ \texttt{then}\ v_3\ \texttt{else}\ \perp
\end{array}
$$

In the equations above, d_i is an element from $\mathcal{D}$ corresponding to the syntactic element c_i and $[\![p_i]\!]$ is the function of signature $(\mathcal{D}^n \to \mathcal{D}_\perp)$ which corresponds to the syntactic symbol p_i.

Functional applications, basic operations $[\![p_i]\!]$ and the cond function have to be monotonous in order to have a well-defined semantics because $\mathcal{D}_\perp$ is simple.

Moreover, basic operations and function calls must be strict: if one of their arguments is $\perp$, the result of the call must be $\perp$. Except for the basic conditional operation if for which we must not evaluate all arguments before the call.

The program aims at giving a meaning to the different functions it contains. Thus, the semantics of a program is a particular functional environment which verifies all the equations of its user-defined functions. We take the least one, so we compute the least fix-point of a transformation of the functional environment Φ. We call $\mathcal{P}$ the function which evaluates the meaning of a program. This one has the following signature:

$$\mathcal{P}[\![prog]\!] : \Phi$$

The meaning of a program is the least fix-point defined by:

$$\begin{array}{lll} \mathcal{P}[\![& f_1(x_1,\ldots,x_n) = expr_1 & \\ & f_2(x_1,\ldots,x_n) = expr_2 & \\ & \vdots & \\ & f_p(x_1,\ldots,x_n) = expr_p \;\;]\!] & = \; \mathsf{fix}(\lambda\phi.\langle \; \mathcal{E}[\![expr_1]\!]\;\phi, \\ & & \quad\; \mathcal{E}[\![expr_2]\!]\;\phi, \\ & & \quad\; \vdots \\ & & \quad\; \mathcal{E}[\![expr_p]\!]\;\phi\rangle) \end{array}$$

where fix is the function defined in chapter 1 section 1.4. Each function f_i is bound to a function ϕ_i of signature $\mathcal{D}^n \rightarrow \mathcal{D}_\perp$ such that the meaning of a program is the least fix-point of $\phi = \langle \mathcal{E}[\![expr_1]\!]\;\phi,\ldots,\mathcal{E}[\![expr_p]\!]\;\phi\rangle$. This definition is correct if and only if Φ is a domain and $\lambda\phi.\langle \mathcal{E}[\![expr_1]\!]\;\phi,\ldots,\mathcal{E}[\![expr_p]\!]\;\phi\rangle$ is continuous. Φ is effectively a domain following the function domain definition mentioned in chapter 1 section 1.5.3 and $\lambda\phi.\langle \mathcal{E}[\![expr_1]\!]\;\phi,\ldots,\mathcal{E}[\![expr_p]\!]\;\phi\rangle$ is continous.

In order to implement the fix-point computation, a function called `p` is defined A.1. Within a convention representation, this function has the same signature than the semantic function $\mathcal{P}$. The problem is to compute the fix-point of $\mathcal{P}[\![prog]\!]$, thus of `p prog`. This fix-point verifies the equation `p prog phi = phi`. But we may not write this equation in `caml`. On the other hand, we may write `phi f x = p prog phi f x`. This expression is mathematically equivalent to `p prog phi = phi` according the function equality definition described in [Sto77]. Note that all fix-point verifies this equation. But `caml` will compute the least one because its semantics is strict.

The reader may find in appendix A.1 an implementation of this semantics.

2.3 A call-by-name interpreter

Now, on the basis of the first implementation, we define the semantics of a call-by-name interpreter. A call-by-name interpreter evaluates arguments of a function call only if necessary. How can we change the semantics of the call-by-value to obtain an interpreter which implements the call-by-name ? The purpose of this section is to answer this question.

We have to revisit the variable and the functional environment. A variable or an argument of a function call may be $\perp$ because arguments are not evaluated immediately or may be never evaluated. In some cases, if an argument of a function call is undefined, its evaluation may terminate. Hence, we have the following definitions:

$$\begin{array}{lll} \rho & \in & \mathcal{D}_\perp^n \\ \phi & \in & \Phi \equiv (\mathcal{D}_\perp^n \rightarrow \mathcal{D}_\perp)^p \end{array}$$

The signature of the expression evaluation function has the same form as in the call-by-value case:

$$\mathcal{E}[\![expr]\!] : \Phi \rightarrow \mathcal{D}_{\perp}^{n} \rightarrow \mathcal{D}_{\perp}$$

The semantics of expressions is almost the same as in the call-by-value case. The only difference resides in the evaluation of user-defined function calls. In the call-by-name case, this evaluation does not have to be strict because an argument could be $\perp$. We have the following equation:

$$\mathcal{E}[\![f_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho = \phi_i(\mathcal{E}[\![expr_1]\!]\ \phi\ \rho, \ldots, \mathcal{E}[\![expr_n]\!]\ \phi\ \rho)$$

The function $\mathcal{P}$ which computes the meaning of a program *prog* does not change.

The reader can find in appendix A.2 an implementation of this semantics. Note that the implementation of the interpreter is written in `caml` which is a language using strict evaluation. We know that an interpeter using call-by-name evaluates arguments of function calls as late as possible. How could we delay the evaluation of the arguments in a strict interpreter? To delay the evaluation of an argument, we are using the rule of abstraction-reduction described in [Sto77] [Rou97]. The reduction rule allows us to replace in the body of a function all occurences of the real parameter by the value of the formal parameter. On the other hand, the abstraction rule allows us to put in a formal parameter and to put out a real parameter. If the expression $f[v/x]$ means "substitute the variable x by the expression v in the body of the function f". The abstraction-reduction rule may be described as $(\lambda x.f(x))v \iff f[v/x]$.

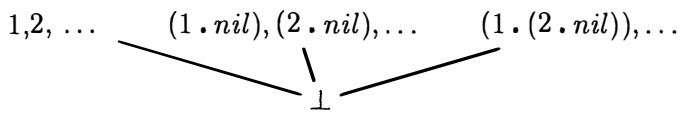
When $f(e)$ is computed, e is evaluated first. We may apply the abstraction rule to the argument e: $(\lambda x.e)x$. This expression is evaluated to e when the function is applied to an argument of any type. We are free to choose the type of the argument. In the implementation, we choose the unit value `()`. Hence, we have that e is equivalent to $(\lambda x.e)()$. The function $f = \lambda y.expr$ is therefore transformed into another function: $f' = \lambda y.expr[z/y]$ with $z = y()$. The application $f'(\lambda().e)$ is therefore delayed. Indeed, the evaluated argument is a function which gives a result only when applied to a value of unit type.

2.4 A call-by-value interpreter with lists

Now, we examine how to upgrade our first descriptions such that the interpreter supports *lists*. By definition, a list is a sequence of ordered values of the same type for typed lists, of any types for non-typed lists. We consider non-typed lists. The set of values containing this kind of list is $\mathcal{D}^*$ which is defined as follows:

$$\mathcal{D}^* = \mathcal{D} \cup (\mathcal{D}^* \times \mathcal{D}^*)$$

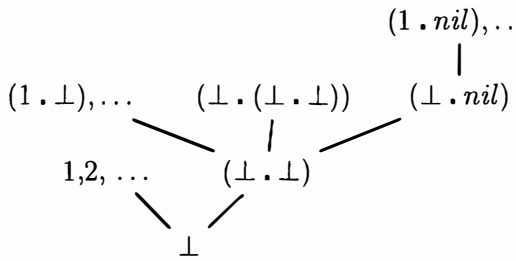
Elements from $\mathcal{D}^*$ are either written as $1, 2, \ldots$ for atoms[1] or $(\ldots)$ for pairs. Moreover, in $\mathcal{D}$ is added the element corresponding to the empty list (noted `nil`). On the basis of this set, we can construct the lifted domain $(\mathcal{D}^*)_\perp$ which is a flat domain described as follows:



This domain contains well-defined lists (without $\perp$ as component) and has $\perp$ as lower bound. A second interesting domain is $(\mathcal{D}_\perp)^*$ which is constructed from $\mathcal{D}_\perp$ and contains lazy lists. A lazy list is a list which may contain one or more $\perp$ inside. This domain is defined as follows:

$$(D_\perp)^* = \mathcal{D}_\perp \cup ((\mathcal{D}_\perp)^* \times (\mathcal{D}_\perp)^*)$$

The structure of this domain is quite different from the previous one. We have a tree structure which looks like the following:



In a call-by-value interpreter, lists are defined in the domain $(\mathcal{D}^*)_\perp$. On the other hand, in a call-by-name interpreter, lists are defined in the domain $(\mathcal{D}_\perp)^*$ because an argument may never be completely evaluated during the execution of a program.

In the call-by-value interpreter with lists, a variable and function call argument can be a simple value from $\mathcal{D}$ or a list from $\mathcal{D}^* \times \mathcal{D}^*$. The result of a function call must be an element from $(\mathcal{D}^*)_\perp$. Therefore, the variable and the functional environments are modified as follows:

$$\begin{aligned} \rho &\in \mathcal{D}^{*n} \\ \phi &\in \Phi \equiv (\mathcal{D}^{*n} \rightarrow (\mathcal{D}^*)_\perp)^p \end{aligned}$$

The expression evaluation function $\mathcal{E}$ always needs as arguments: the variable and the functional environment. Hence, the signature of the expression evaluation becomes:

$$\mathcal{E}[\![expr]\!] : \Phi \rightarrow \mathcal{D}^{*n} \rightarrow (\mathcal{D}^*)_\perp$$

[1] If $\mathcal{D}$ contains integers.

With respect to the semantics defined in section 2.2, we add the evaluation of the three common primitives: `cons`, `car` and `cdr`.

$$\mathcal{E}[\![\underline{\texttt{cons}}(expr_1, expr_2)]\!]\ \phi\ \rho = \texttt{strict cons}(\mathcal{E}[\![expr_1]\!]\ \phi\ \rho, \mathcal{E}[\![expr_2]\!]\ \phi\ \rho)$$
$$\mathcal{E}[\![\underline{\texttt{cdr}}(expr)]\!]\ \phi\ \rho = \texttt{strict cdr}(\mathcal{E}[\![expr]\!]\ \phi\ \rho)$$
$$\mathcal{E}[\![\underline{\texttt{car}}(expr)]\!]\ \phi\ \rho = \texttt{strict car}(\mathcal{E}[\![expr]\!]\ \phi\ \rho)$$

where `strict` is defined as in section 2.2 and `cons`, `car` and `cdr` are defined as follows:

Given h, t in $\mathcal{D}^*$, we have:

$$\texttt{cons}\ (h, t) = (h\,.\,t)$$

Given x in $\mathcal{D}^*$, we have:

$$\begin{array}{ll} \texttt{cdr}\ x = t & \text{if } x = (h\,.\,t) \\ \texttt{cdr}\ x = \bot & \text{if } x \in \mathcal{D} \end{array}$$

Given x in $\mathcal{D}^*$, we have:

$$\begin{array}{ll} \texttt{car}\ x = h & \text{if } x = (h\,.\,t) \\ \texttt{car}\ x = \bot & \text{if } x \in \mathcal{D} \end{array}$$

We have that `cond`, `car` and `cdr` are continous on $(\mathcal{D}^*)_\bot$. The function $\mathcal{P}$ which computes the meaning of a program *prog* does not change anymore.

The reader can find in appendix A.3 an implementation of this semantics.

2.5 A call-by-name interpreter with lazy lists

In this section, we present a call-by-name interpreter with lazy lists. The nature of a non-strict language is that their data constructors are also non-strict. Non-strict constructors permit the definition of (conceptually) infinite data structures. An example of a program which generates infinite lists is described in section 2.6.

With regard to the semantics, we use the domain $(\mathcal{D}_\bot)^*$ described in section 2.4 to define the variable and the functional environments:

$$\begin{array}{lll} \rho & \in & (\mathcal{D}_\bot)^{*n} \\ \phi & \in & \Phi \equiv ((\mathcal{D}_\bot)^{*n} \to (\mathcal{D}_\bot)^*)^p \end{array}$$

Hence, the signature of the expression evaluation function becomes:

$$\mathcal{E}[\![expr]\!] : \Phi \to (\mathcal{D}_\bot)^{*n} \to (\mathcal{D}_\bot)^*$$

With respect to the interpreter described in section 2.3, we only add the evaluation of the three primitives for lists manipulations. Because an

element of a list may not be evaluated, these functions do not have to be strict.

$$\mathcal{E}[\![\underline{\texttt{cons}}(expr_1, expr_2)]\!]\ \phi\ \rho = \texttt{cons}(\mathcal{E}[\![expr_1]\!]\ \phi\ \rho, \mathcal{E}[\![expr_2]\!]\ \phi\ \rho)$$
$$\mathcal{E}[\![\underline{\texttt{cdr}}(expr)]\!]\ \phi\ \rho = \texttt{cdr}(\mathcal{E}[\![expr]\!]\ \phi\ \rho)$$
$$\mathcal{E}[\![\underline{\texttt{car}}(expr)]\!]\ \phi\ \rho = \texttt{car}(\mathcal{E}[\![expr]\!]\ \phi\ \rho)$$

where `cons`, `car`, `cdr` are defined as in section 2.4.

We are always using the same $\mathcal{P}$ function in order to evaluate the meaning of a program.

The first order language that we have developped until now is the one we use to study strictness analysis in section 2.7.

The reader can find in appendix A.4 an implementation of this semantics.

2.6 Examples

For our examples, we assume that our language has the following built-in primitives and $\mathcal{D}$ is the set of integers:

`pred` *expr*	: computes the predecessor of *expr*.
`succ` *expr*	: computes the successor of *expr*.
`if` $(expr_1, expr_2, expr_3)$	: conditional command: if the evaluation of $expr_1$ is 0 then returns the evaluation of $expr_3$ otherwise returns the evaluation of $expr_2$.
`equ` $(expr_1, expr_2)$	: returns 1 if the evaluation of both expressions $expr_1$ and $expr_2$ is equal, 0 otherwise.

In order to simplify examples, we also assume that the following user-defined functions are written:

`times` (x, y)	: returns $x * y$.
`plus` (x, y)	: returns $x + y$.

Call-by-name is time-consuming An example of program may be the following:

```
fact x = if( equ(x,0),
             1,
             times( x, fact (pred x)))
fibo x = if( equ(x,0),
             1,
             if( equ(n,1),
                 1,
                 plus( fibo (pred x), fibo (pred (pred x)))
               )
           )
```

This program shows us the efficiency of the call-by-value with respect to call-by-name when we are calling `fibo(fact 4)` for example. The reason

resides in the fact that call-by-name does not evaluate arguments immediatly: arguments are evaluated as late as possible. Table 2.1 shows how the time-consuming the call-by-name interpreter is[2].

Call	Name	Value
`fibo(fact(1))`	0.00	0.00
`fibo(fact(2))`	0.01	0.00
`fibo(fact(3))`	72.15	0.00
`fibo(fact(4))`	-	0.01

Table 2.1: Computation time in seconds

Interest of lazy lists Another interesting example is a program which generates infinite lists and returns the first n elements.

```
listnum x = cons( x, listnum (succ x))
firstn (x, lst) = if( eqn(x, 0),
                      nil,
                      if( atom(lst),
                          nil,
                          cons( car(lst),
                                firstn( pred x, cdr(lst)))
                        )
                      )
```

If we perform the call `firstn(10, (listnum 0))`, a strict evaluator does not terminate because it computes `listnum 0` which generates an infinite list. In the other hand, a non-strict evaluator gives us the right answer because, during the execution, the expression `listnum 0` is never actually evaluated.

2.7 Fix-point computation using minimal function graph

2.7.1 Motivations

In the call-by-value case, computing the fix-point as described in the previous sections may fail to terminate. For a function call to $h(v)$ where $h(x) = h(x)$ and $v \in \mathcal{D}$, a strict evaluator does not terminate[3]. As we describe further, minimal function graph is another way of computing the fix-point

[2]Tests were made on a Celeron 700Mhz with 64MiB RAM

[3]Note that a non-strict interpreter does not terminate either

of a program. With this method, computing the fix-point of $h(v)$ terminates and gives the value $\perp$ as result. On the other hand, for a function call to $g(v)$ where $g(x) = g(x-1)$ and $v \in \mathcal{D}$, the evaluation does not terminate either. A strictness interpreter using minimal function graph gives us a solution to our termination problem.

2.7.2 Minimal function graph

Minimal function graph (hereafter abbreviated *mfg*) is an approach which consists in computing fix-point evaluating arguments of a function call only when they are needed. The semantics groups together intermediate calls during the evaluation of a program. In this approach, the idea is to describe functions as a set of arguments-result pairs in order to identify these functions. From a set of arguments-result pairs, the mfg approach gives the least set of pairs that are needed in order to compute the result. In this semantics, function calls are represented as *closures*: a pair $(f_i, (v_1, \ldots, v_n))$ where f_i is the function name and the tuple $(v_1, \ldots, v_n)$ is a list of arguments. Furthermore, we distinguish two sets: a set $\mathcal{C}$ which collects the different closures appearing in the computation and a set Φ which is the functional environment defined as a Cartesian product instead of a function. We call $\mathcal{C}$ a need-set. To explain how this approach works, consider the simple example of the *factorial* function defined as:

$$\underline{\texttt{let}}\ fact(n) = \underline{\texttt{if}}\ n = 1\ \underline{\texttt{then}}\ 1\ \underline{\texttt{else}}\ n * fact(n-1)$$

Let us examine now the function call to *fact* with the argument 3. During the execution, we check first if 3 is equal to 1. Since this is not the case, we compute the "*else branch*" of the conditional command. At this stage, we don't know the value of $fact(3)$, hence $fact(3)$ is mapped to the undefined value $\perp$. We also see that $fact(3)$ requires that the value of $fact(2)$ be computed. Hence, the set $\mathcal{C}$ contains the tuples $(fact, 3)$ and $(fact, 2)$. The next iteration consists in computing the value of $fact(2)$. With the same reasoning, $fact(2)$ is mapped to $\perp$ and we see that in order to compute it we need the value of $fact(1)$. The pair $(fact, 1)$ is added to the set $\mathcal{C}$. The computation of $fact(1)$ returns directly the value 1. The knowledge of this value leads us to compute the value of $fact(2)$ and later, the value of $fact(3)$. Finally, the set $\mathcal{C}$ contains the pairs $\{(fact, 3), (fact, 2), (fact, 1)\}$. We obtain a graph which results from the computation of different values. Note that we have computed only arguments that are needed.

2.7.3 An interpreter using mfg

As we have seen in the previous section, in order to write the semantics of an interpreter using mfg algorithm, we have to define an additional set. This

set called $\mathcal{C}$ captures arguments of function calls that are needed to evaluate the initial call. $\mathcal{C}$ is a p-uple of sets:

$$\mathbf{c} \in \mathcal{C} \equiv \wp(\mathcal{D}^{*n})^p$$

The variable environment does not change with respect to the call-by-value interpreter:

$$\rho \in \mathcal{D}^{*n}$$

We redefine Φ as a p-uple which contains sets of argument values and the result of function calls with these arguments:

$$\phi \in \Phi \equiv (\wp(\mathcal{D}^{*n} \times (\mathcal{D}^*)_\perp))^p$$

The expression evaluation function does not really change compared to the call-by-value interpreter with lists. We have the following signature for the expression evaluation function:

$$\mathcal{E}[\![expr]\!] : \Phi \dashrightarrow \mathcal{D}^{*n} \to (\mathcal{D}^*)_\perp$$

and the semantics of expressions is very close to the one of the call-by-value interpreter:

$$\begin{array}{l}
\mathcal{E}[\![c_i]\!]\ \phi\ \rho = d_i \\
\mathcal{E}[\![x_i]\!]\ \phi\ \rho = \rho_i \\
\mathcal{E}[\![p_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho = \\
\qquad \texttt{strict}\ [\![p_i]\!](\mathcal{E}[\![expr_1]\!]\ \phi\ \rho, \ldots, \mathcal{E}[\![expr_n]\!]\ \phi\ \rho) \\
\mathcal{E}[\![\underline{\texttt{if}}(expr_1, expr_2, expr_3)]\!]\ \phi\ \rho = \\
\qquad \texttt{cond}(\mathcal{E}[\![expr_1]\!]\ \phi\ \rho, \mathcal{E}[\![expr_2]\!]\ \phi\ \rho, \mathcal{E}[\![expr_3]\!]\ \phi\ \rho) \\
\mathcal{E}[\![f_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho = \texttt{strict}\ \bar{\phi}_i(\mathcal{E}[\![expr_1]\!]\ \phi\ \rho, \ldots, \mathcal{E}[\![expr_n]\!]\ \phi\ \rho)
\end{array}$$

where `strict` and `cond` are defined as in section 2.2 and where $\bar{\phi}_i$ is the function that the tabulated function ϕ_i represents. Given ϕ_i a tabulate function from ϕ, we note:

$$\begin{array}{rccl}
\bar{\phi}_i : & \mathcal{D}^{*n} & \to & (\mathcal{D}^*)_\perp \\
& \langle d_1, \ldots, d_n \rangle & \rightsquigarrow & \begin{cases} d_{n+1} & \text{if } \langle d_1, \ldots, d_n, d_{n+1} \rangle \in \phi_i, \\ \perp & \text{otherwise.} \end{cases}
\end{array}$$

Now, we define a function $\mathcal{N}$ which collects arguments of a function call that need to be evaluated. This function takes an expression, a functional environment and a variable environment, and it constructs as a result an element from $\mathcal{C}$:

$$\mathcal{N}[\![expr]\!] : \Phi \dashrightarrow \mathcal{D}^{*n} \dashrightarrow \mathcal{C}$$

The semantics of $\mathcal{N}$ is the following:

$$\mathcal{N}[\![c_i]\!]\ \phi\ \rho = \langle \emptyset, \ldots, \emptyset \rangle$$
$$\mathcal{N}[\![x_i]\!]\ \phi\ \rho = \langle \emptyset, \ldots, \emptyset \rangle$$
$$\mathcal{N}[\![p_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho = (\mathcal{N}[\![expr_1]\!]\ \phi\ \rho) \otimes^{\cup} \ldots \otimes^{\cup} (\mathcal{N}[\![expr_n]\!]\ \phi\ \rho)$$
$$\mathcal{N}[\![\underline{\texttt{if}}(expr_1, expr_2, expr_3)]\!]\ \phi\ \rho =$$
$$\texttt{condn}(expr_1, expr_2, expr_3)$$
$$\mathcal{N}[\![f_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho =$$
$$\texttt{mkset}(\ (\mathcal{E}[\![expr_1]\!]\ \phi\ \rho), \ldots, (\mathcal{E}[\![expr_n]\!]\ \phi\ \rho), i)$$
$$\otimes^{\cup}\ (\mathcal{N}[\![expr_1]\!]\ \phi\ \rho) \otimes^{\cup} \ldots \otimes^{\cup} (\mathcal{N}[\![expr_n]\!]\ \phi\ \rho)$$

where

$$\begin{array}{l}
\texttt{condn}\ (expr_1, expr_2, expr_3) = \\
\quad \texttt{if}\ (\mathcal{E}[\![expr_1]\!]\ \phi\ \rho)\ \texttt{then} \\
\qquad (\mathcal{N}[\![expr_1]\!]\ \phi\ \rho) \otimes^{\cup} (\mathcal{N}[\![expr_2]\!]\ \phi\ \rho) \\
\quad \texttt{else} \\
\qquad \texttt{if not}\ (\mathcal{E}[\![expr_1]\!]\ \phi\ \rho)\ \texttt{then} \\
\qquad\quad (\mathcal{N}[\![expr_1]\!]\ \phi\ \rho) \otimes^{\cup} (\mathcal{N}[\![expr_3]\!]\ \phi\ \rho) \\
\qquad \texttt{else}\ (\mathcal{N}[\![expr_1]\!]\ \phi\ \rho)
\end{array}$$

and

$$\begin{array}{l}
\texttt{mkset}\ (v_1, \ldots, v_n, i) = \\
\quad \texttt{if}\ (v_1 = \perp \vee \ldots \vee v_n = \perp)\ \texttt{then}\ \langle \emptyset, \ldots, \emptyset \rangle \\
\quad \texttt{else}\ \langle \emptyset, \ldots, \emptyset, \{\langle v_1, \ldots, v_n \rangle\}_i, \emptyset, \ldots, \emptyset \rangle
\end{array}$$

and

$$\begin{array}{lrcl}
\otimes^{\cup}: & \mathcal{C} \times \mathcal{C} & \to & \mathcal{C} \\
\langle\ \langle \mathbf{c}_1, \ldots, \mathbf{c}_p \rangle, \langle \mathbf{c}'_1, \ldots, \mathbf{c}'_p \rangle\ \rangle & & \rightsquigarrow & \langle \mathbf{c}_1, \ldots, \mathbf{c}_p \rangle \otimes^{\cup} \langle \mathbf{c}'_1, \ldots, \mathbf{c}'_p \rangle = \\
 & & & \langle \mathbf{c}_1 \cup \mathbf{c}'_1, \ldots, \mathbf{c}_p \cup \mathbf{c}'_p \rangle
\end{array}$$

The operator $\otimes^{\cup}$ guarantee us the monotony of the $\mathcal{N}$ function and is associative since the union of sets is associative.

In the capture of function calls for user-defined functions, if one of the arguments is $\perp$, then we don't add this function call to the result of the evaluation. However, we add function calls that are needed in order to evaluate each argument.

The fixpoint computation is slightly different of the previous ones because we compute two different sets: Φ which represents the graph of a function and $\mathcal{C}$ which contains the captures of function calls. We define a function $\mathcal{P}$ which compute the meaning of a program. As we need to know

which inital calls are performed at the start, $\mathcal{P}$ takes an extra argument $\mathbf{c}_0$ which contains them. Its signature is the following:

$$\mathcal{P}[\![prog]\!]\ \mathbf{c}_0 : \Phi \times \mathcal{C}$$

The meaning of a program is then the least fix-point defined as follows:

$$\mathcal{P}[\![prog]\!]\ \mathbf{c}_0 = \mathsf{fix}(\lambda\langle\phi, \mathbf{c}\rangle.\langle\phi \otimes^{\sqcup} (\texttt{iterate}\ prog\ \phi\ \mathbf{c}), \mathbf{c} \otimes^{\cup} (\texttt{needs}\ prog\ \phi\ \mathbf{c})\rangle)$$

where `iterate` should compute a new graph by calling the expression evaluation function $\mathcal{E}$ for all calls in $\mathcal{C}$ and `needs` should collect all needs using the $\mathcal{N}$ function for all calls in $\mathcal{C}$. The function $\otimes^{\sqcup}$ is defined in the same way as the $\otimes^{\cup}$ operator. Instead of computing the union of its arguments, it computes the least upper bound and it is defined on Φ. The fix-point is well-defined because the upper bound operator will ensure us that the function is monotonic.

Functions `iterate` and `needs` are defined as follows:

$$\begin{array}{l}
\texttt{iterate}\ [\![\ \ f_1(x_1, \ldots, x_n) = expr_1 \\
\qquad\qquad f_2(x_1, \ldots, x_n) = expr_2 \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad f_p(x_1, \ldots, x_n) = expr_p\ \ ]\!]\ \phi\ \langle \mathbf{c}_1, \ldots, \mathbf{c}_p\rangle = \\
\quad \langle\ \ \{\langle\rho, \mathcal{E}[\![expr_1]\!]\ \phi\ \rho\ \mid\ \rho \in \mathbf{c}_1\rangle\} \\
\qquad\qquad\qquad , \ldots , \\
\qquad \{\langle\rho, \mathcal{E}[\![expr_p]\!]\ \phi\ \rho\ \mid\ \rho \in \mathbf{c}_p\rangle\}\ \ \rangle
\end{array}$$

$$\begin{array}{l}
\texttt{needs}\ [\![\ \ f_1(x_1, \ldots, x_n) = expr_1 \\
\qquad\qquad f_2(x_1, \ldots, x_n) = expr_2 \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad f_p(x_1, \ldots, x_n) = expr_p\ \ ]\!]\ \phi\ \langle \mathbf{c}_1, \ldots, \mathbf{c}_p\rangle = \\
\qquad\qquad (\otimes^{\cup}_{\rho \in \mathbf{c}_1} \mathcal{N}[\![expr_1]\!]\ \phi\ \rho) \\
\qquad\qquad\qquad\qquad \otimes^{\cup} \ldots \otimes^{\cup} \\
\qquad\qquad (\otimes^{\cup}_{\rho \in \mathbf{c}_p} \mathcal{N}[\![expr_p]\!]\ \phi\ \rho)
\end{array}$$

The reader can find an implementation of this semantics in appendix A.5.

Example

For this example, let's take the Fibonnaci series (see code in section 2.6). We call the function `fibo` with the argument 3. This is an interesting example because this function is doubly recursive.

First, we begin with an empty functional environment and a need-set ($\mathcal{C}$) which is initialized with the arguments of the initial call ($fibo, 3$).

$$\begin{array}{|l|}\hline \phi \parallel \emptyset \\ \hline \mathcal{C} \parallel \{(fibo, 3)\} \\ \hline \end{array}$$

The conditional command is evaluated. As x is not equal to 0 nor to 1, the *else branch* of the conditional command is computed. The expression `plus( fibo (pred x), fibo (pred (pred x)))` is evaluated. When this expression is evaluated we see that it is not possible to return a result for $fibo(3)$. Hence, the functional environment is upgraded with the tuple $(fibo, 3, \perp)$. During the evaluation of the previous expression, the first argument of `plus` is evaluated first. We notice here that we have to compute $fibo(2)$ to get value of the initial call. Thus, the need-set is upgraded with the tuple $(fibo, 2)$.

$$\begin{array}{|l|}\hline \phi \parallel (fibo, 3, \perp) \\ \hline \mathcal{C} \parallel \{(fibo, 3), (fibo, 2)\} \\ \hline \end{array}$$

The next step consists of computing $fibo(2)$ which comes from the previous call. As the functional environment does not contain the result of the call to $fibo(2)$, it is evaluated. As x is not equal to 0 or to 1, the *else branch* of the conditional command is evaluated. The expression `plus( fibo (pred x), fibo (pred (pred x)))` is once more evaluated. When this expression is evaluated we see that it is not possible to return a result for $fibo(2)$. Hence, the functional environment is upgraded with the tuple $(fibo, 2, \perp)$. During the evaluation of the previous expression, the first argument of `plus` is evaluated first. We may notice that we have to compute $fibo(1)$ to get the value of $fibo(3)$. The need-set becomes:

$$\begin{array}{|l|}\hline \phi \parallel (fibo, 3, \perp), (fibo, 2, \perp) \\ \hline \mathcal{C} \parallel \{(fibo, 3), (fibo, 2), (fibo, 1)\} \\ \hline \end{array}$$

Now, $fibo(1)$ which comes from the previous call to $fibo(2)$ is evaluated. As the functional environment does not contain the result of the call to $fibo(1)$, it is evaluated. As x is equal to 1, the function call returns 1. The functional environment is then upgraded with the tuple $(fibo, 1, 1)$. We obtain the following sets:

$$\begin{array}{|l|}\hline \phi \parallel (fibo, 3, \perp), (fibo, 2, \perp), (fibo, 1, 1) \\ \hline \mathcal{C} \parallel \{(fibo, 3), (fibo, 2), (fibo, 1)\} \\ \hline \end{array}$$

We come back to the previous call. Now the second argument of `plus` is computed. As x has the value 2, the expression that has to be computed is $fibo(0)$. First we check if we have not already computed the value of this call. As this is not the case, the call is evaluated and returns as result the value 1. The need-set is upgraded with the tuple $(fibo, 0)$. These results allow us to evaluate the value of $fibo(2)$ and we return back to the first call to `plus`.

$\phi \parallel (fibo, 3, \bot), (fibo, 2, 2), (fibo, 1, 1), (fibo, 0, 1)$
$\mathcal{C} \parallel \{(fibo, 3), (fibo, 2), (fibo, 1)(fibo, 0)\}$

Now the second argument of `plus` of the first call is computed. Recall that x has the value 3. Thus the expression that has to be computed is $fibo(1)$. As the value of this expression is already known, we can return a result for the initial call. We upgrade once more the functional environment and we get the result.

$\phi \parallel (fibo, 3, 3), (fibo, 2, 2), (fibo, 1, 1), (fibo, 0, 1)$
$\mathcal{C} \parallel \{(fibo, 3), (fibo, 2), (fibo, 1)(fibo, 0)\}$

Note that in this example, in order to simplify notation and reduce space-consumption, we have taken care to not directly express all function calls. Normaly, ϕ and $\mathcal{C}$ must contain occurences of function calls to `plus`, `equ` and `if`.

2.7.4 A strictness interpreter using mfg

In order to define a strictness interpretation, we use a two-point domain that we note **2**. Its elements are 0 and 1 and are ordered by $0 \sqsubseteq 1$.

$$\mathbf{2} = \{0, 1\}, \qquad 0 \sqsubseteq 1$$

This domain is defined to describe whether an element in $(\mathcal{D}^*)$ is defined or not. The $\bot$ element is mapped to 0 and other values are mapped to 1. For this purpose we can define an abstraction function:

$$\alpha : (\mathcal{D}^*) \to \mathbf{2}$$
$$\alpha(d) = \texttt{if } d = \bot \texttt{ then 0 else 1}$$

On the basis of this domain **2**, we can easily redefine the semantics of the previous section (2.7.3). We just substitute all $\mathcal{D}^*$'s in the previous semantics. The variable and the functional environments are defined as follow:

$$\begin{aligned} \rho &\in \mathbf{2}^n \\ \phi &\in \Phi \equiv \wp(\mathbf{2}^n \times \mathbf{2})^p \end{aligned}$$

In order to ensure the safety properties, we need that the function evaluation of expression verifies the following equations:

$$\begin{aligned}
&\alpha(\mathcal{E}[\![c_i]\!]\ \phi\ \rho) \sqsubseteq \mathcal{E}^\sharp[\![c_i]\!]\ \phi\ \rho \\
&\alpha(\mathcal{E}[\![x_i]\!]\ \phi\ \rho) \sqsubseteq \mathcal{E}^\sharp[\![x_1]\!]\ \phi\ \rho \\
&\alpha(\mathcal{E}[\![p_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho) \\
&\qquad \sqsubseteq \mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \rho \wedge \ldots \wedge \mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \rho \\
&\alpha(\mathcal{E}[\![\underline{\texttt{if}}(expr_1, expr_2, expr_3)]\!]\ \phi\ \rho \\
&\qquad \sqsubseteq \mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \rho \wedge (\mathcal{E}^\sharp[\![expr_2]\!]\ \phi\ \rho \vee \mathcal{E}^\sharp[\![expr_3]\!]\ \phi\ \rho) \\
&\alpha(\mathcal{E}[\![f_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho) \\
&\qquad \sqsubseteq \phi_i(\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \rho, \ldots, \mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \rho)
\end{aligned}$$

In this way, we examine strictness properties of expressions. For basic operations and function call, if one of their arguments is undefined (0), their result is undefined (0). For the conditional expression, if $expr_1$ is undefined we know that the result is undefined, otherwise, it is either the result of the evaluation of $expr_2$ or the result of the evaluation of $expr_3$. The result returned by $\mathcal{E}^\sharp$ is an upper bound, hence if the upper bound is the least element 0 then we know that the evaluation of the expression by $\mathcal{E}$ is undefined.

We mark the semantic function with a sharp ($\sharp$) in order to distinguish it from the previous function. The signature of the expression evaluation function is straightforward:

$$\mathcal{E}^\sharp[\![expr]\!] : \Phi \to \mathbf{2}^n \to \mathbf{2}$$

and we have the following semantics:

$$\begin{array}{l}
\mathcal{E}^\sharp[\![c_i]\!]\ \phi\ \rho = 1 \\
\mathcal{E}^\sharp[\![x_i]\!]\ \phi\ \rho = \rho_i \\
\mathcal{E}^\sharp[\![p_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho = (\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \rho) \wedge \ldots \wedge (\mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \rho) \\
\mathcal{E}^\sharp[\![\underline{\texttt{if}}(expr_1, expr_2, expr_3)]\!]\ \phi\ \rho = \\
\qquad\qquad (\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \rho) \wedge ((\mathcal{E}^\sharp[\![expr_2]\!]\ \phi\ \rho) \vee (\mathcal{E}^\sharp[\![expr_3]\!]\ \phi\ \rho)) \\
\mathcal{E}^\sharp[\![f_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho = \bar{\phi}_i(\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \rho, \ldots, \mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \rho)
\end{array}$$

For the need function, the set $\mathcal{C}$ is defined similarly:

$$\mathbf{c} \in \mathcal{C} \equiv \wp(\mathbf{2}^n)^p$$

The signature of the need function becomes:

$$\mathcal{N}^\sharp[\![expr]\!] : \Phi \to \mathbf{2}^n \to \mathcal{C}$$

And the associated need function $\mathcal{N}$ is also straightforward:

$$\begin{array}{l}
\mathcal{N}^\sharp[\![c_i]\!]\ \phi\ \rho = \langle \emptyset, \ldots, \emptyset \rangle \\
\mathcal{N}^\sharp[\![x_i]\!]\ \phi\ \rho = \langle \emptyset, \ldots, \emptyset \rangle \\
\mathcal{N}^\sharp[\![p_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho = (\mathcal{N}^\sharp[\![expr_1]\!]\ \phi\ \rho) \otimes^\cup \ldots \otimes^\cup (\mathcal{N}^\sharp[\![expr_n]\!]\ \phi\ \rho) \\
\mathcal{N}^\sharp[\![f_i^\sharp(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho = \\
\qquad \texttt{mkset}(\ (\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \rho), \ldots, (\mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \rho), i) \\
\qquad \otimes^\cup (\mathcal{N}^\sharp[\![expr_1]\!]\ \phi\ \rho) \otimes^\cup \ldots \otimes^\cup (\mathcal{N}^\sharp[\![expr_n]\!]\ \phi\ \rho)
\end{array}$$

where functions `mkset` and $\otimes^\cup$ are defined like in section 2.7.3. The function which computes the fix-point is the same as this described in the same section.

2.8 Example

Examine functions h where `h(x) = h(x)` and g where `g(x) = g(x-1)`. The table 2.2 shows us the result of function calls in the different cases. A sign - means that function calls do not terminate; $\perp$ that the execution of the algorithm returns the value $\perp$ and 0 the value 0 of the set **2**.

	`h(x) = h(x)`	`g(x) = g(x-1)`
Call-by-value	-	-
Call-by-name	-	-
mfg	$\perp$	-
strictness mfg	0	0

Table 2.2: Function results

How do we interpret these results ? In the call-by-value and the call-by-name cases, the interpreter does not terminate. The mfg is an improvment of the fix-point computation, but it does not terminate in many cases either. A strictness interpreter using mfg working on a finite domain terminates in all cases.

The aim of strictness analysis is to transform call-by-name into call-by-value when results of function calls are identical. In order to do that, we study the strictness properties of user-defined functions. It means that if the evaluation of an argument of a function call does not terminate then the evaluation of the function call does not terminate either. When we work on a finite set such as **2**, the evaluation of a function call becomes independent of the parameter passing mode and we are sure to obtain a result. Now, let us take back the example with lazy lists in section 2.6 page 25. The table 2.3 shows the different results of the strictness interpreter with calls to the function `firstn`.

Call	**Results**
`firstn (1, 0)`	1
`firstn (0, 1)`	0

Table 2.3: Strictness of `firstn`.

What do these results mean ? They mean that the function `listnum` is strict in its first argument but not in the second one. Hence, we know that we are able transform call-by-name for the first argument into call-by-value. That is what we expected.

In the example in section 2.6 page 24, the study of strictness of `fact x` and `fibo x` allow us to know that we may replace call-by-name by call-by-value. Hence the computation time of the result should be better.

Chapter 3

Second-order case

3.1 Introduction

This chapter is devoted to the study of strictness analysis on a second order functional language. In chapter 2, we have given a succinct description of the first order case without giving real proofs of what we have done. The reader interested in a complete developement of the first order case may read [Pol96]. We dress in this chapter a more complete development of the second order case.

3.2 A second order call-by-value interpreter

In this section, we are interested in the study of a second order call-by-value functional language. First, we dress its mathematical semantics, then we set out its caml implementation.

3.2.1 Mathematical semantics

Syntactic sets

We define syntactic sets containing symbols which allow us to write the abstract syntax of the language.

c	$\in \mathbb{C}ons$	:	Set of constants (or basic symbols)
x	$\in \mathbb{V}arv$	:	Set of variables
h	$\in \mathbb{V}arf$	:	Set of first order functional variables
f	$\in \mathbb{F}unc_1$	:	Set of first order functional constants
g	$\in \mathbb{F}unc_2$	:	Set of second order functional constants
p	$\in \mathbb{P}rim$	:	Set of primitives (first order predefined functions)
$expr$	$\in \mathbb{E}xpr$	:	Set of expressions
$prog$	$\in \mathbb{P}rog$	:	Set of programs

Abstract syntax

A functional language is essentially made up of expressions. An expression is either a constant (c) or a variable (x_i) or a basic operation ($p_i(expr_1, \ldots, expr_n)$) or a call to a first order user-defined function ($f_i(expr_1, \ldots, expr_n)$). Moreover, we distinguish two second order calls. The first one with a parameter name h ($g_i(expr_1, \ldots, expr_n)h$) and the second one with a first order function f_j ($g_i(expr_1, \ldots, expr_n)f_j$) as functional argument. The last call corresponds to the evaluation of the parameter name with its arguments ($h(expr_1, \ldots, expr_n)$).

$$
\begin{array}{rcl}
expr & ::= & c \\
 & | & x_i \\
 & | & p_i(expr_1, \ldots, expr_n) \\
 & | & f_i(expr_1, \ldots, expr_n) \\
 & | & g_i(expr_1, \ldots, expr_n)h \\
 & | & g_i(expr_1, \ldots, expr_n)f_j \\
 & | & h(expr_1, \ldots, expr_n)
\end{array}
$$

A program is composed of first and second order user-defined functions descriptions:

$$
\begin{array}{rcl}
prog & ::= & f_1(x_1, \ldots, x_n) = expr_1 \\
 & | & f_2(x_1, \ldots, x_n) = expr_2 \\
 & \vdots & \vdots \\
 & | & f_p(x_1, \ldots, x_n) = expr_p \\
 & | & g_1(x_1, \ldots, x_n)h = expr'_1 \\
 & | & g_2(x_1, \ldots, x_n)h = expr'_2 \\
 & \vdots & \vdots \\
 & | & g_q(x_1, \ldots, x_n)h = expr'_q
\end{array}
$$

We point out that like in the first order case, all functions possess the same number of arguments. We can replace non-used arguments and functions by dummy variables to solve the problem. Moreover, second order user-defined functions are defined with only one functional variable (h). If we want to use more than one h function in a second order call, we have to define a selection function which given a number returns the appropriate h function. The syntax described above does not allow to write functions that returns a function as result: in other words, curryfication is not possible. An example is given in the next section.

Some examples

We assume that our language has basic operations and the first order user-defined functions `plus` and `times` defined in section 2.6.

A first example consists in computing the sum of squares of the n first integers. For this one, we define a first order function `sqr` which, given an integer, computes the square of a number; a second order function `sum` which, given an integer `n` and a first order function `f`, computes `f n + sum (n-1) f` if `n` $\neq$ `0` and returns 0 otherwise; and a first order function `sumsqr`, which given an integer, calls `sum` with this integer and the function `sqr`.

We have:

```
sqr x = times(x,x)
sum n f = if( equ(n,0), 0, plus(f n, sum (pred n) f))
sumsqr n = sum n sqr
```

We may compare our example with the abstract syntax and identify the different function calls:

`times(x,x)`	$f_i(expr_1, \ldots, expr_n)$
`f n`	$h(expr_1, \ldots, expr_n)$
`sum (pred n) f`	$g_i(expr_1, \ldots, expr_n)h$
`sum n sqr`	$g_i(expr_1, \ldots, expr_n)f_j$

Another example shows the usage of the selection function. We define two first order functions `sqr` and `cube` that compute respectively the square and the cube of an integer; a second order function `bigsum` which, given two functions `f` and `g` and an integer `n`, computes `f n + g n + sum (n-1) f g` if `n` $\neq$ `0` and returns 0 otherwise. It is clear that the syntax described above does not allow us to write the following program:

```
sqr x = times(x,x)
cube x = times(times(x,x),x)
bigsum n f g = if( equ(n,0),
                   0,
                   plus(f n, plus(g n, bigsum (pred n) f g)))
sumsqrcube n = bigsum n sqr cube
```

because the second order function `bigsum` takes two first order functional symbols.

In order to respect the syntax, we define a selection function `sel` which given an integer and a list of arguments returns the function call. Hence we have:

```
sqr x = times(x,x)
cube x = times(times(x,x),x)
sel (n,x) = if( equ(n,0),
                sqr x,
                if( equ(n,1),
                    cube x,
                    0 ))
```

```
bigsum n sel = if( equ(n,0),
                   0,
                   plus(sel (0,n),
                        plus(sel (1,n), bigsum (pred n) sel)))
sumsqrcube n = bigsum n sel
```

Values set

We work with a set of values $\mathcal{D}$ which is a general set that can contain simple values as defined in section 2.2 and we define the domain $\mathcal{D}_\perp = \mathcal{D} \cup \{\perp\}$.

Environments

We also define some environments. The first one is a variable environment which does not change with respect to the first order case. Hence, we have:

$$\rho \in \mathcal{D}^n$$

Speaking about the functional environment, we need to define an additional element η which represents the value of the formal parameter h passed to a second order function. Its value is a function from $\mathcal{D}^n$ to $\mathcal{D}_\perp$. Hence, η and ϕ are defined as follows:

η	$\in$	$E \equiv (\mathcal{D}^n \dashrightarrow \mathcal{D}_\perp)$	Parameter value "environment"
ϕ	$\in$	$\Phi \equiv (E \rightarrow \mathcal{D}^n \dashrightarrow \mathcal{D}_\perp)^{p+q}$	Second order functional environment

This definition of the functional environment Φ allows us to group first order and second order functions. In the case of a first order function, the functional parameter h does not appear. Thus, we have to use a trick for η in order to compute the semantics of a first order function. For this purpose, we rely on a function defined as follows:

$$\eta_\perp = \lambda x.\perp \qquad \forall x \in \mathcal{D}^n$$

Note that we could have divided the functional environment into two distinct domains: one that contains first order functions definitions and another that contains second order functions definitions. We have not chosen this representation because it is more complex to compute fixpoint on two domains than one and because we may use the fix-point definition from the previous chapter.

Expression evaluation function

The expression evaluation function takes as arguments a variable environment, a first and a second order functional environment and a parameter value environment. We have the following signature:

$$\mathcal{E}[\![expr]\!] : \Phi \dashrightarrow E \dashrightarrow \mathcal{D}^n \dashrightarrow \mathcal{D}_\perp$$

Each expression is evaluated as follows:

$$
\begin{aligned}
&\mathcal{E}[\![c]\!]\ \phi\ \eta\ \rho = d\\
&\mathcal{E}[\![x_i]\!]\ \phi\ \eta\ \rho = \rho_i\\
&\mathcal{E}[\![p_i(expr_1,\ldots,expr_n)]\!]\ \phi\ \eta\ \rho =\\
&\qquad \texttt{strict}\ [\![p_i]\!](\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho,\ldots,\mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho)\\
&\mathcal{E}[\![\underline{\texttt{if}}(expr_1,expr_2,expr_3)]\!]\ \phi\ \eta\ \rho =\\
&\qquad \texttt{cond}(\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho,\mathcal{E}[\![expr_2]\!]\ \phi\ \eta\ \rho,\mathcal{E}[\![expr_3]\!]\ \phi\ \eta\ \rho)\\
&\mathcal{E}[\![f_i(expr_1,\ldots,expr_n)]\!]\ \phi\ \eta\ \rho =\\
&\qquad \texttt{strict}\ (\phi_i\ \eta_\perp)(\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho,\ldots,\mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho)\\
&\mathcal{E}[\![g_i(expr_1,\ldots,expr_n)h]\!]\ \phi\ \eta\ \rho =\\
&\qquad \texttt{strict}\ (\phi_{p+i}\ \eta)(\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho,\ldots,\mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho)\\
&\qquad\quad \text{if } h \text{ is a parameter name (formal parameter)}\\
&\mathcal{E}[\![g_i(expr_1,\ldots,expr_n)f_j]\!]\ \phi\ \eta\ \rho =\\
&\qquad \texttt{strict}\ (\phi_{p+i}(\phi_j\ \eta_\perp))(\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho,\ldots,\mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho)\\
&\qquad\quad \text{if the parameter value is the named}\\
&\qquad\quad \text{function } f_j \text{(actual parameter)}\\
&\mathcal{E}[\![h(expr_1,\ldots,expr_n)]\!]\ \phi\ \eta\ \rho =\\
&\qquad \texttt{strict}\ \eta(\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho,\ldots,\mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho)
\end{aligned}
$$

where `strict` and `cond` are defined in section 2.2.

In the equations above, d is an element from $\mathcal{D}$ corresponding to the syntactic element c and $[\![p_i]\!]$ is the function of signature $(\mathcal{D}^n \to \mathcal{D}_\perp)$ which corresponds to the syntactic symbol p_i. $(\phi_j\ \eta_\perp)$ is a first order function corresponding to the syntactic element f_j and whose signature is $(\mathcal{D}^n \to \mathcal{D}_\perp)$. Moreover, basic operations, function call and `cond` function have to be monotonous in order to have a well-defined semantics.

Program evaluation function

We call $\mathcal{P}$, the function that evaluates the meaning of a program. It has the same signature as in the first order case:

$$\mathcal{P}[\![prog]\!] : \Phi$$

The meaning of a program is the least fix-point defined by:

$$
\begin{array}{lll}
\mathcal{P}[\![& f_1(x_1,\ldots,x_n) = expr_1 & \\
& f_2(x_1,\ldots,x_n) = expr_2 & \\
& \vdots & \\
& f_p(x_1,\ldots,x_n) = expr_p & \\
& g_1(x_1,\ldots,x_n)h = expr'_1 & \\
& g_2(x_1,\ldots,x_n)h = expr'_2 & \\
& \vdots & \\
& g_q(x_1,\ldots,x_n)h = expr'_q \quad]\!] & = \quad \mathsf{fix}(\lambda\phi.\langle \quad \mathcal{E}[\![expr_1]\!]\ \phi, \\
& & \qquad\qquad \mathcal{E}[\![expr_2]\!]\ \phi, \\
& & \qquad\qquad \vdots \\
& & \qquad\qquad \mathcal{E}[\![expr_p]\!]\ \phi, \\
& & \qquad\qquad \mathcal{E}[\![expr'_1]\!]\ \phi, \\
& & \qquad\qquad \mathcal{E}[\![expr'_2]\!]\ \phi, \\
& & \qquad\qquad \vdots \\
& & \qquad\qquad \mathcal{E}[\![expr'_q]\!]\ \phi\rangle)
\end{array}
$$

where fix is the function defined in chapter 1 section 1.4. This definition is correct if and only if Φ is a domain and $\lambda\phi.\langle\mathcal{E}[\![expr_1]\!]\ \phi,\ldots,\mathcal{E}[\![expr_n]\!]\ \phi,\mathcal{E}[\![expr'_1]\!]\ \phi,\ldots,\mathcal{E}[\![expr'_n]\!]\ \phi\rangle$ is continuous. Φ is effectively a domain following the function domain definition mentioned in chapter 1 section 1.5.3.

3.2.2 A caml implementation

Before writing a `caml` interpreter for this second order call-by-value functional language, we have to give an instantiation of the abstract syntax described in the beginning of this section. First, we instantiate $\mathcal{D}$ as the union of integers and strings:

$$\mathcal{D} = \mathbb{Z} \cup \mathbb{S}$$

We define primitives p_i: **succ**, **pred**, **equ**, **if** and **cat** which are respectively the successor, the predecessor, the equality test, the conditional

command and the concatenation of strings. The syntax is the following:

$$
\begin{array}{rcl}
expr & ::= & d_i \\
& | & x_i \\
& | & \mathbf{succ}(expr) \\
& | & \mathbf{pred}(expr) \\
& | & \mathbf{equ}(expr_1, expr_2) \\
& | & \mathbf{if}(expr_1, expr_2, expr_3) \\
& | & \mathbf{cat}(expr_1, expr_2) \\
& | & f_i(expr_1, \ldots, expr_n) \\
& | & g_i(expr_1, \ldots, expr_n)h \\
& | & g_i(expr_1, \ldots, expr_n)f_j \\
& | & h(expr_1, \ldots, expr_n)
\end{array}
$$

Primitives $[\![p_i]\!]$ are specified as follows:

$$
\begin{array}{rrcll}
succ: & \mathcal{D} & \to & \mathcal{D}_\perp & \\
 & n & \rightsquigarrow & n+1 & \text{if } n \in \mathbb{Z} \\
pred: & \mathcal{D} & \to & \mathcal{D}_\perp & \\
 & n & \rightsquigarrow & n-1 & \text{if } n \in \mathbb{Z} \\
cat: & \mathcal{D} \times \mathcal{D} & \to & \mathcal{D}_\perp & \\
 & (n_1, n_2) & \rightsquigarrow & n_1 \hat{} n_2 & \text{if } n_1, n_2 \in \mathbb{S} \\
equ: & \mathcal{D} \times \mathcal{D} & \to & \mathcal{D}_\perp & \\
 & (n, n) & \rightsquigarrow & 1 & \text{if } n \in \mathcal{D} \\
 & (n_1, n_2) & \rightsquigarrow & 0 & \text{if } n_1, n_2 \in \mathbb{Z} \text{ and } n_1 \neq n_2 \\
 & (n_1, n_2) & \rightsquigarrow & 0 & \text{if } n_1, n_2 \in \mathbb{S} \text{ and } n_1 \neq n_2 \\
if: & \mathcal{D} \times \mathcal{D} \times \mathcal{D} & \to & \mathcal{D}_\perp & \\
 & (0, n_1, n_2) & \rightsquigarrow & n_2 & \\
 & (n, n_1, n_2) & \rightsquigarrow & n_1 & \text{if } n \in \mathbb{Z} \setminus \{0\}
\end{array}
$$

The translation of the abstract syntax to `caml` is rather straightforward. The set of values $\mathcal{D}$ which contains integers and strings is translated into a `caml` object of type `val`:

```
type val = I of int | S of string ;;
```

Elements from $\mathbb{E}xpr$ are translated into `caml` object of type `expr`:

```
type expr = INT of int
          | STR of string
          | VAR of string
          | SUCC of expr
          | PRED of expr
          | EQU of expr*expr
          | IF of expr*expr*expr
          | CAT of expr*expr
          | CALL of string*(expr list)*string;;
```

An element of $\mathbb{P}rog$ is translated to a `caml` object of the form:

```
type prog = PROG of (string*(string list)*string*expr) list;;
```

where components of tuples are respectively the function name, the list of variable names, the functional argument name and an expression which is the body of the function.

In order to specify correctly `caml` functions corresponding to semantic functions, we introduce a *meta-function* which maps an object from the mathematical world to an object from the `caml` world. We will note it $(.)^c$ [Pol96]. So, if x is an element of $\mathbb{V}arv$, we note its representation x^c and we write abusively $(.)^c : \mathbb{V}arv \rightarrow$ `string`. From now on, we have:

$$\begin{aligned}
(.)^c &: \mathbb{V}arv \rightarrow \texttt{string},\\
(.)^c &: \mathbb{V}arf \rightarrow \texttt{string},\\
(.)^c &: \mathbb{E}xpr \rightarrow \texttt{expr},\\
(.)^c &: \mathbb{P}rog \rightarrow \texttt{prog},\\
(.)^c &: \mathbb{F}unc_1 \rightarrow \texttt{string},\\
(.)^c &: \mathbb{F}unc_2 \rightarrow \texttt{string}.
\end{aligned}$$

Primitives are translated into `caml` functions below which are a direct transcription from their specification above:

```
let succ = fun (I n) -> (I (n+1))
         |       _   -> raise (NotAValidType "in succ function");;

let pred = fun (I n) -> (I (n-1))
         |       _   -> raise (NotAValidType "in pred function");;

let equ = fun (I i , I j) -> if (i=j) then (I 1) else (I 0)
        |     (S s , S t) -> if (s=t) then (I 1) else (I 0)
        |     ( _  ,  _ ) -> raise (NotAValidType "in equ function");;

let cat = fun (S s , S t) -> S (s^t)
        |     ( _  ,  _ ) -> raise (NotAValidType "in cat function");;

let myif = fun (I 0, x, y) -> y
         |     (I n, x, y) -> x
         |     ( _ , x, y) -> raise (NotAValidType "in myif function");;
```

As `caml` use call-by-value, we don't call the `myif` function defined above in order to implement the conditional command. If we call this function, `caml` will evaluate the three arguments before returning a result and in the case of non-termination, this result may be invalid. We have to use the `if then else` of `caml` which uses lazy evaluation.

We can specify primitives, expressions and a program as follows.

- For primitives:

$$\forall n \in \mathcal{D} \quad \texttt{succ}\ n^c = (succ(n))^c$$
$$\forall n \in \mathcal{D} \quad \texttt{pred}\ n^c = (pred(n))^c$$
$$\forall n_1, n_2 \in \mathcal{D} \quad \texttt{cat}(n_1^c, n_2^c) = (cat(n_1, n_2))^c$$
$$\forall n_1, n_2 \in \mathcal{D} \quad \texttt{equ}(n_1^c, n_2^c) = (equ(n_1, n_2))^c$$

- For expressions:

$$\forall c \in \mathbb{C}ons \quad (c)^c = \texttt{INT}\ d \qquad \text{if } d \in \mathbb{Z}$$
$$\forall c \in \mathbb{C}ons \quad (c)^c = \texttt{STR}\ d \qquad \text{if } d \in \mathbb{S}$$
$$\forall x_i \in \mathbb{V}arv \quad (x_i)^c = \texttt{VAR}\ x_i$$

where x_i is the string which represents the name of the variable

$$\forall expr \in \mathbb{E}xpr \quad (\textbf{succ}\ expr)^c = \texttt{SUCC}\ expr^c$$
$$\forall expr \in \mathbb{E}xpr \quad (\textbf{pred}\ expr)^c = \texttt{PRED}\ expr^c$$
$$\forall expr_1, expr_2 \in \mathbb{E}xpr \quad (\textbf{equ}\ (expr_1, expr_2))^c = \texttt{EQU}\ (expr_1^c, expr_2^c)$$
$$\forall expr_1, expr_2 \in \mathbb{E}xpr \quad (\textbf{cat}\ (expr_1, expr_2))^c = \texttt{CAT}\ (expr_1^c, expr_2^c)$$
$$\forall expr_1, expr_2, expr_3 \in \mathbb{E}xpr \quad (\textbf{if}\ (expr_1, expr_2, expr_3))^c = \texttt{IF}\ (expr_1^c, expr_2^c, expr_3^c)$$
$$\forall expr_1, \ldots, expr_n \in \mathbb{E}xpr\ \forall f_i \in \mathbb{F}unc_1 \quad (f_i(expr_1, \ldots, expr_n))^c = \texttt{CALL}(f_i^c, [\ expr_1^c\ ; \ldots ;\ expr_n^c\ ], '')$$
$$\forall expr_1, \ldots, expr_n \in \mathbb{E}xpr\ \forall g_i \in \mathbb{F}unc_2\ \forall h \in \mathbb{V}arf \quad (g_i(expr_1, \ldots, expr_n)h)^c = \texttt{CALL}(g_i^c, [\ expr_1^c\ ; \ldots ;\ expr_n^c\ ], h^c)$$
$$\forall expr_1, \ldots, expr_n \in \mathbb{E}xpr\ \forall g_i \in \mathbb{F}unc_2\ \forall f_j \in \mathbb{F}unc_1 \quad (g_i(expr_1, \ldots, expr_n)f_j)^c = \texttt{CALL}(g_i^c, [\ expr_1^c\ ; \ldots ;\ expr_n^c\ ], f_j^c)$$
$$\forall expr_1, \ldots, expr_n \in \mathbb{E}xpr\ \forall h \in \mathbb{V}arf \quad (h(expr_1, \ldots, expr_n))^c = \texttt{CALL}(h^c, [\ expr_1^c\ ; \ldots ;\ expr_n^c\ ], '')$$

- For programs

$$\forall prog \in \mathbb{P}rog \quad (prog)^c = \texttt{PROG}\ [\ (f_1^c, [\ x_1^c\ ; \ldots ;\ x_n^c\ ], expr_1^c) \ ;\ \ldots\ ; \ (f_p^c, [\ x_1^c\ ; \ldots ;\ x_n^c\ ], expr_p^c) \ (g_1^c, [\ x_1^c\ ; \ldots ;\ x_n^c\ ], expr_1'^c) \ ;\ \ldots\ ; \ (g_q^c, [\ x_1^c\ ; \ldots\ ; x_n^c\ ], expr_q'^c)]$$

Now, the next step is to define the `caml` type associated to the variable and the functional environments. An object of the type `rho` implements an element of ρ and an object of the type `phi` is an element of Φ.

```
type rho = RHO of (string*val) list;;

type phi = PHI of (string -> (val list) -> string -> val) ;;
```

where components of tuples from `rho` are respectively a variable's name and a variable's value and `phi` is a function which takes a function name, a list of argument's values, a parameter name and returns a value which is the result of the function.

We have:

$$(.)^c : \Phi \rightarrow \texttt{phi}$$
$$(.)^c : \rho \rightarrow \texttt{rho}$$

We also can specify the variable and functional environments as follows:

$$\forall \rho \in \mathcal{D}^n \quad (\rho)^c = [\ (x_1^c, \rho_1^c)\ ;\ \ldots\ ;\ (x_n^c, \rho_n^c)\ ]$$
$$\forall \phi \in (E \rightarrow \mathcal{D}^n \rightarrow \mathcal{D}_\perp)^{p+q}$$
$$\text{let } \phi^c = (\texttt{PHI}\ \varphi) \text{ we have}$$
$$\text{if } i \leq p \quad (\phi_i \eta_\perp \rho)^c = \varphi f_i^c \rho^c s \qquad \text{where} s \text{ is anything}$$
$$\text{if } i > p \quad (\phi_i(\varphi f_j)\rho)^c = \varphi g_i^c \rho^c f_j^c$$

Now that we have defined `caml` types and primitives, we can construct the evaluation function of expressions. The construction of this function is a simple pattern matching upon the structure of expressions. We call this function `e`. To take advantage of curryfication we modify the signature order:

$$\mathcal{E}[\![expr]\!] : \Phi \rightarrow E \rightarrow \mathcal{D}^n \rightarrow \mathcal{D}_\perp$$

becomes

$$\texttt{e} : \texttt{rho} \rightarrow \texttt{phi} \rightarrow \texttt{string} \rightarrow \texttt{string} \rightarrow \texttt{expr} \rightarrow \texttt{val}$$

In order to avoid redundancy in the functional environment, the parameter value is defined as a string which references an element from ϕ.

Note that the `caml` function represents correctly the semantic function $\mathcal{E}$:

$$(\mathcal{E}[\![expr]\!]\phi(\phi f_i)\rho)^c = \texttt{e}\ \rho^c \phi^c h^c f_i^c expr^c$$

The translation of the expression evaluation function $\mathcal{E}$ into the `caml` function `e` is straightforward except for the function call for which we give an explanation. In the semantics we have four cases of function calls: two first order function calls (call to f_i and h) and two second order function calls (both with g_i but with a different parameter). If the function name `f` is different from the parameter name, we know that `f` is a second order function. Thus, we apply the function returned by (`phi f`) to the result of the evaluation of the list of expressions and either f_i if `s` is the parameter name `pn` or h which is `s` otherwise. The case where f is equal to `pn` corresponds to the first order case.

```
let rec e (RHO rho) (PHI phi) pn pv =
 fun (INT i)        -> (I i)
 |   (STR s)        -> (S s)
 |   (VAR s)        -> valofs s rho
 |   (CALL (f,l,s)) ->
  if (f = pn)
  then (phi pv) (map (e (RHO rho) (PHI phi) pn pv) l) (lookupf s pn pv)
  else (phi f) (map (e (RHO rho) (PHI phi) pn pv) l) (lookupf s pn pv)
 |   (IF (c, a, b)) ->
  if (istrue (e (RHO rho) (PHI phi) pn pv c))
  then (e (RHO rho) (PHI phi) pn pv a)
  else (e (RHO rho) (PHI phi) pn pv b)
 |   (SUCC x)     -> succ (e (RHO rho) (PHI phi) pn pv x)
 |   (PRED x)     -> pred (e (RHO rho) (PHI phi) pn pv x)
 |   (CAT (x,y))  ->
       cat ( (e (RHO rho) (PHI phi) pn pv x),
             (e (RHO rho) (PHI phi) pn pv y) )
 |   (EQU (x,y))  ->
       equ ( (e (RHO rho) (PHI phi) pn pv x),
             (e (RHO rho) (PHI phi) pn pv y) );;
```

As explained in chapter 2, the fix-point is computed with a function called p. This function has the following signature:

$$\texttt{p} : \texttt{prog} \rightarrow \texttt{phi} \rightarrow \texttt{string} \rightarrow \texttt{val list} \rightarrow \texttt{string} \rightarrow \texttt{val}$$

We may specify this function in terms of representation convention:

$$\mathcal{P}[\![prog]\!] = \texttt{p}\ prog^c \phi^c (\phi f_j)^e [\ x_1^c; \ldots; x_n^c\ ] h^c$$

The fix-point computation is implemented in caml as follows:

```
let rec p = fun (PROG []) (PHI phi) f vl pv -> raise (EmptyList "in prog")
          |     (PROG ((fn, pl, q, expr)::r)) (PHI phi) f vl pv ->
                  if (f=fn)
                  then (e (RHO (zip pl vl)) (PHI phi) q pv expr)
                  else p (PROG r) (PHI phi) f vl pv;;

let rec fixphy (PROG prog) f vl =
        p (PROG prog) (PHI (fixphy (PROG prog))) f vl;;
```

The reader can find the complete `caml` implementation in appendix B.1.

3.3 A second order call-by-name interpreter

3.3.1 Mathematical semantics

Now, we examine how to implement a call-by-name interpreter on the basis of the call-by-value interpreter. The syntax of the language does not change: we are taking the one defined in section 3.2.1.

Environments

In call-by-name, variables and arguments of a function call may be $\perp$ as explained in section 2.3. Hence variable and functional environments are defined as follows:

ρ	$\in$	$\mathcal{D}_\perp^n$	Variable environment
η	$\in$	$E \equiv (\mathcal{D}_\perp^n \to \mathcal{D}_\perp)$	Parameter value "environment"
ϕ	$\in$	$\Phi \equiv (E \to \mathcal{D}_\perp^n \to \mathcal{D}_\perp)^{p+q}$	Second order functional environment

Expression evaluation function

The signature of the expression evaluation function has the following signature:

$$\mathcal{E}[\![expr]\!] : \Phi \to E \to \mathcal{D}_\perp^n \to \mathcal{D}_\perp$$

The only difference with the call-by-value case is that function calls are not strict. Other expressions are evaluated in the same way.

$$\begin{aligned}
&\mathcal{E}[\![f_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \eta\ \rho = \\
&\qquad (\phi_i\ \eta_\perp)(\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho, \ldots, \mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho) \\
&\mathcal{E}[\![g_i(expr_1, \ldots, expr_n)h]\!]\ \phi\ \eta\ \rho = \\
&\qquad (\phi_{p+i}\ \eta)(\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho, \ldots, \mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho) \\
&\qquad\qquad \text{if } h \text{ is a parameter name (formal parameter)} \\
&\mathcal{E}[\![g_i(expr_1, \ldots, expr_n)f_j]\!]\ \phi\ \eta\ \rho = \\
&\qquad (\phi_{p+i}(\phi_j\ \eta_\perp))(\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho, \ldots, \mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho) \\
&\qquad\qquad \text{if the parameter value is the named} \\
&\qquad\qquad \text{function } f_j \text{(actual parameter)} \\
&\mathcal{E}[\![h(expr_1, \ldots, expr_n)]\!]\ \phi\ \eta\ \rho = \\
&\qquad \eta(\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho, \ldots, \mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho)
\end{aligned}$$

where $\eta_\perp$ is defined in section 3.2.1.

Program evaluation function

The program evaluation function is defined as above and possesses the same signature.

3.3.2 A caml implementation

For a discussion of the implementation, please consult section 3.5.2 which considers this language extended with lazy lists. The reader can find a `caml` implementation in appendix B.2.

3.4 A second order call-by-value interpreter with lists

In this section, we upgrade types by adding lists to simple types. The syntax of the language does not change.

3.4.1 Mathematical semantics

Values set

The set of values we use is the same as defined in section 2.4:

$$\mathcal{D}^* = \mathcal{D} \cup (\mathcal{D}^* \times \mathcal{D}^*)$$

and we use the domain $(\mathcal{D}^*)_\perp$ as defined in section 2.4 in order to define the functional environment.

Environments

A variable is a simple value from $\mathcal{D}$ or a list from the set $\mathcal{D}^* \times \mathcal{D}^*$. Hence the variable environment is defined as follows:

$$\rho \in \mathcal{D}^{*n}$$

Arguments of a function call take their values in the set $\mathcal{D}^*$ because an argument of a function can be a simple value or a list. A function call returns a value from $(\mathcal{D}^*)_\perp$. Hence we have these definitions:

η	$\in$	$E \equiv (\mathcal{D}^{*n} \to (\mathcal{D}^*)_\perp)$	Parameter value "environment"
ϕ	$\in$	$\Phi \equiv (E \to \mathcal{D}^{*n} \to (\mathcal{D}^*)_\perp)^{p+q}$	Second order functional environment

Expressions evaluation function

In order to evaluate expressions, we define a function which takes a variable environment, a parameter environment and a second order functional environment and returns a value of $(\mathcal{D}^*)_\perp$. Hence, the signature of the expression evaluation function becomes:

$$\mathcal{E}[\![expr]\!] : \Phi \to E \to \mathcal{D}^{*n} \to (\mathcal{D}^*)_\perp$$

As in the first order case and with respect to the semantics of a second order call-by-value interpreter defined in section 3.2.1, the three common primitives `cons`, `car` and `cdr` are added to the set of primitives. Their definitions are given in section 2.4.

For expressions evaluation, we add the evaluation of the three operators on lists to the one of the call-by-value interpreter described earlier in this chapter (see section 3.2.1). The evaluation of these three primitives is the same as the one described in section 2.4.

Program evaluation function

The function $\mathcal{P}$ which computes the meaning of a program *prog* does not change anymore.

3.4.2 A caml implementation

For the `caml` implementation, we change the type `val`, which is the `caml` representation of $\mathcal{D}$, so that it supports lists.

```
type val = I of int
         | S of string
         | P of val*val
         | NULL ;;
```

In the set of expressions we add the three primivites on lists and we specify them as follows:

$$
\begin{array}{rcl}
expr & ::= & \ldots \\
 & | & \mathbf{car}(expr) \\
 & | & \mathbf{cdr}(expr) \\
 & | & \mathbf{cons}(expr_1, expr_2)
\end{array}
$$

$$
\begin{array}{rrcll}
car: & \mathcal{D}^* & \to & (\mathcal{D}^*)_\perp & \\
 & lst & \rightsquigarrow & h & \text{if } lst = (h \,.\, t) \\
cdr: & \mathcal{D}^* & \to & (\mathcal{D}^*)_\perp & \\
 & lst & \rightsquigarrow & t & \text{if } lst = (h \,.\, t) \\
cons: & \mathcal{D}^* \times \mathcal{D}^* & \to & (\mathcal{D}^*)_\perp & \\
 & (h, t) & \rightsquigarrow & (h \,.\, t) &
\end{array}
$$

These basic operations are translated into the `caml` functions below which are a direct transcription of their specification above.

```
let car = fun (P (x,y)) -> x
        |         _     -> raise (NotAPair "in car function");;

let cdr = fun (P (x,y)) -> y
        |         _     -> raise (NotAPair "in cdr function");;

let cons = fun x  y  -> P (x , y);;
```

In terms of representation function, the `caml` implementation corresponds accurately to the definition of the semantics.

$$
\begin{array}{l}
\forall expr \in \mathbb{E}xpr \quad (\mathbf{car}\ expr)^c = \texttt{CAR}\ expr^c \\
\forall expr \in \mathbb{E}xpr \quad (\mathbf{cdr}\ expr)^c = \texttt{CDR}\ expr^c \\
\forall expr_1, expr_2 \in \mathbb{E}xpr \quad (\mathbf{cons}\ (expr_1, expr_2))^c = \texttt{CONS}\ (expr_1^c, expr_2^c)
\end{array}
$$

And we have:

$$\forall l \in \mathcal{D}^* \quad \texttt{car}\ l^c = (car(l))^c$$
$$\forall l \in \mathcal{D}^* \quad \texttt{cdr}\ l^c = (cdr(l))^c$$
$$\forall h, l \in \mathcal{D}^* \quad \texttt{cons}\ (h^c, l^c) = (cons(h, l))^c$$

The reader interested by a complete implementation can find the `caml` code in appendix B.3.

3.5 A second order call-by-name interpreter with lists

3.5.1 Mathematical semantics

The definition of a second order call-by-name interpreter with lists is really straightforward on the basis of the previous work.

Environments

The value of a variable is a simple value from the set $\mathcal{D}_\perp$ or a list. Lists may contain $\perp$. Hence, if the value of a variable is a list, this value comes from $(\mathcal{D}_\perp)^*$. The result of a function call or a parameter value (which is a function) comes from $(\mathcal{D}_\perp)^*$.

Thus, variable and functional environments are defined as follows:

ρ	$\in$	$(\mathcal{D}_\perp)^{*n}$	Variable environment
η	$\in$	$E \equiv ((\mathcal{D}_\perp)^{*n} \to (\mathcal{D}_\perp)^*)$	Parameter value "environment"
ϕ	$\in$	$\Phi \equiv (E \to (\mathcal{D}_\perp)^{*n} \to (\mathcal{D}_\perp)^*)^{p+q}$	Second order functional enironment

Expression evaluation function

The evaluation function of expressions takes as arguments the expression that must be evaluated, a functional environment, a parameter value "environment" and a variable environment. This function gives as result an element of $(\mathcal{D}_\perp)^*$. Its signature is the following:

$$\mathcal{E}[\![expr]\!] : \Phi \dashrightarrow E \to (\mathcal{D}_\perp)^{*n} \dashrightarrow (\mathcal{D}_\perp)^*$$

Equations for expression evaluation functions are drawn from the first order call-by-name interpreter with list 2.5 for primitives and the second order call-by-name interpreter 3.3 for function calls.

The function $\mathcal{P}$ which computes the meaning of a program *prog* does not change anymore.

3.5.2 A caml implementation

`Caml` is a strict language. We have already explained in section 2.3 how to simulate call-by-name in a language that implements call-by-value. Since we would like that arguments be evaluated as late as possible, we redefine the `caml` type which represents $\mathcal{D}^*$.

```
type val = I of int
         | S of string
         | P of nval*nval
         | NULL
and nval ==  (unit -> val) ;;
```

`nval` is the type on which the interpreter works. As lazy lists may contain undefined elements (see section 2.5), basic operations for their construction/destruction have to be defined on `nval`. On the other hand, other primivites are defined on `val` because they have to be strict. Hence, **cons**, **car** and **cdr** are written in `caml` as follows:

```
let cons = fun x  y  -> myout (P (x , y));;

let car x = match myin x
            with (P(x,y))  -> x
             |       _     -> raise (NotAPair "in car function");;

let cdr x = match myin x
            with  (P(x,y)) -> y
             |       _      -> raise (NotAPair "in cdr function");;
```

where `myin` and `myout` are respectively the reduction and the abstraction rules (explained in section 2.3). These functions have the following signature:

$$\texttt{myin} : (() \rightarrow \texttt{val}) \rightarrow \texttt{val}$$
$$\texttt{myout} : \texttt{val} \rightarrow (() \rightarrow \texttt{val})$$

Their respective `caml` code is:

```
let myin a = a ()

let myout a () = a
```

The reader interested by the implementation can find it in appendix B.4.

3.6 Example

Suppose that we have the basic operations and user-defined function defined in the section 2.6. A second order program working on list may be the following:

```
fact x = if( equ(x,0),
             1,
             times( x, fact (pred x)))
fibo x = if( equ(x,0),
             1,
             if( equ(n,1),
                 1,
                 plus( fibo (pred x), fibo (pred (pred x)))
               )
           )
fibfact n = fibo (fact n)
map l f = if( (atom l),
              nil,
              cons( f (car l), map (cdr l) f)
            )
mapfibfact l = map l fibfact
```

An example of a call may be `mapfibfact (1;2;3;4)`.

3.7 A second order interpreter using mfg

3.7.1 Introduction

We have already spoken about minimal function graph in section 2.7. Here, we explain how to implement the minimal function graph algorithm for the second order case. The execution of the algorithm is the same as in the first order case. Note however that there is a little difference. In the second order case, the second argument of a second order user-defined function is a function. In order to compute the minimal function graph of this second order user-defined function, we associate the functional argument to the result of the computation of the associated minimal function graph. Our development are based on the one described in section 2.7.3

3.7.2 Mathematical semantics

The definition of the variable environment does not change with respect to the call-by-value interpreter with list described in section 3.4. The parameter and functional environment are not defined as a function any more. The parameter environment is the Cartesian product between variable environment and the domain of function results. Whereas the functional environment is the Cartesian product between parameter name, variable environment and the domain of function result. They are defined as follows:

$\rho \in \mathcal{D}^{*n}$	Variable environment
$\eta \in E \equiv \wp(\mathcal{D}^{*n} \times (\mathcal{D}^*)_\perp)$	Parameter value "environment"
$\phi \in \Phi \equiv \wp(E \times \mathcal{D}^{*n} \times (\mathcal{D}^*)_\perp)^{p+q}$	Functional environment

We define a set $\mathcal{C}$ which has the same role as in the first order case (see section 2.7.3). But its definition is different. We collect tuples of arguments of functions which have to be computed and their associated functional actual parameter (which is the value of the functional formal parameter of the second order function). Hence we have:

$$\mathbf{c} \in \mathcal{C} \equiv \wp(\mathcal{D}^{*n} \times E)^{p+q}$$

The signature of the expression evaluation function is defined as usual:

$$\mathcal{E}[\![expr]\!] : \Phi \dashrightarrow E \dashrightarrow \mathcal{D}^{*n} \dashrightarrow (\mathcal{D}^*)_\perp$$

We have the following evaluations for expressions:

$\mathcal{E}[\![c]\!]\ \phi\ \eta\ \rho = d$
$\mathcal{E}[\![x_i]\!]\ \phi\ \eta\ \rho = \rho_i$
$\mathcal{E}[\![p_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \eta\ \rho =$
$\quad$ `strict` $[\![p_i]\!](\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho, \ldots, \mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho)$
$\mathcal{E}[\![\underline{\texttt{if}}(expr_1, expr_2, expr_3)]\!]\ \phi\ \eta\ \rho =$
$\quad$ `cond`$(\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho, \mathcal{E}[\![expr_2]\!]\ \phi\ \eta\ \rho, \mathcal{E}[\![expr_3]\!]\ \phi\ \eta\ \rho)$
$\mathcal{E}[\![f_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \eta\ \rho =$
$\quad$ `strict` $\bar{\phi}_i\ \bar{\eta_\perp}\ (\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho, \ldots, \mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho)$
$\mathcal{E}[\![g_i(expr_1, \ldots, expr_n)h]\!]\ \phi\ \eta\ \rho =$
$\quad$ `strict` $\bar{\phi}_{p+i}\ \bar{\eta}\ (\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho, \ldots, \mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho)$
$\quad\quad$ if h is a parameter name (formal parameter)
$\mathcal{E}[\![g_i(expr_1, \ldots, expr_n)f_j]\!]\ \phi\ \eta\ \rho =$
$\quad$ `strict` $\bar{\phi}_{p+i}(\bar{\phi}_j\ \bar{\eta_\perp})(\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho, \ldots, \mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho)$
$\quad\quad$ if the parameter value is the named
$\quad\quad$ function f_j(actual parameter)
$\mathcal{E}[\![h(expr_1, \ldots, expr_n)]\!]\ \phi\ \eta\ \rho =$
$\quad$ `strict` $\bar{\eta}\ (\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho, \ldots, \mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho)$

where $\bar{\phi}_i$, $\bar{\eta}$ and $\bar{\eta_\perp}$ are functions that the tabulate functions ϕ_i, η and $\eta_\perp$ represent. They are defined as follows:

$$\begin{array}{lllll} \bar{\phi}_i : & (\mathcal{D}^{*n} \to \mathcal{D}_\perp) & \dashrightarrow & \mathcal{D}^{*n} & \dashrightarrow\ (\mathcal{D}^*)_\perp \\ & \bar{\eta} & \dashrightarrow & \langle d_1, \ldots, d_n \rangle & \rightsquigarrow \begin{cases} d_{n+1} & \text{if } \langle \eta, d_1, \ldots, d_n, d_{n+1} \rangle \in \phi_i, \\ \perp & \text{otherwise.} \end{cases} \end{array}$$

and

$$\begin{array}{lll} \bar{\eta} : & \mathcal{D}^{*n} & \dashrightarrow\ (\mathcal{D}^*)_\perp \\ & \langle d_1, \ldots, d_n \rangle & \rightsquigarrow \begin{cases} d_{n+1} & \text{if } \langle d_1, \ldots, d_n, d_{n+1} \rangle \in \eta, \\ \perp & \text{otherwise.} \end{cases} \end{array}$$

and

$$\begin{array}{lll} \bar{\eta_\perp} : & \mathcal{D}^{*n} & \dashrightarrow\ (\mathcal{D}^*)_\perp \\ & \langle d_1, \ldots, d_n \rangle & \rightsquigarrow\ \perp \end{array}$$

The $\mathcal{N}$ function is a function which collects closures of the different function calls that appear during the evaluation. This function returns a set $\mathcal{C}$. Its signature is the following:

$$\mathcal{N}[\![expr]\!] : \Phi \to E \to \mathcal{D}^{*n} \to \mathcal{C}$$

The calls are collected as follows:

$$\begin{array}{l}
\mathcal{N}[\![c]\!]\ \phi\ \eta\ \rho = \langle \emptyset, \ldots, \emptyset \rangle \\
\mathcal{N}[\![x_i]\!]\ \phi\ \eta\ \rho = \langle \emptyset, \ldots, \emptyset \rangle \\
\mathcal{N}[\![p_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \eta\ \rho = \\
\qquad (\mathcal{N}[\![expr_1]\!]\ \phi\ \eta\ \rho) \otimes^{\cup} \ldots \otimes^{\cup} (\mathcal{N}[\![expr_n]\!]\ \phi\ \eta\ \rho) \\
\mathcal{N}[\![\underline{\texttt{if}}(expr_1, expr_2, expr_3)]\!]\ \phi\ \eta\ \rho = \\
\qquad \texttt{condn}(expr_1, expr_2, expr_3) \\
\mathcal{N}[\![f_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \eta\ \rho = \\
\qquad \texttt{mkset}(\ (\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho), \ldots, (\mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho), \emptyset, i) \\
\qquad\quad \otimes^{\cup} (\mathcal{N}[\![expr_1]\!]\ \phi\ \eta\ \rho) \otimes^{\cup} \ldots \otimes^{\cup} (\mathcal{N}[\![expr_n]\!]\ \phi\ \eta\ \rho) \\
\mathcal{N}[\![g_i(expr_1, \ldots, expr_n)h]\!]\ \phi\ \eta\ \rho = \\
\qquad \texttt{mkset}(\ (\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho), \ldots, (\mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho), \eta, i) \\
\qquad\quad \otimes^{\cup} (\mathcal{N}[\![expr_1]\!]\ \phi\ \eta\ \rho) \otimes^{\cup} \ldots \otimes^{\cup} (\mathcal{N}[\![expr_n]\!]\ \phi\ \eta\ \rho) \\
\qquad\quad \text{if } h \text{ is a parameter name (formal parameter)} \\
\mathcal{N}[\![g_i(expr_1, \ldots, expr_n)f_j]\!]\ \phi\ \eta\ \rho = \\
\qquad \texttt{mkset}(\ (\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho), \ldots, (\mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho), \phi_j, i) \\
\qquad\quad \otimes^{\cup} (\mathcal{N}[\![expr_1]\!]\ \phi\ \eta\ \rho) \otimes^{\cup} \ldots \otimes^{\cup} (\mathcal{N}[\![expr_n]\!]\ \phi\ \eta\ \rho) \\
\qquad\quad \text{if the parameter value is the named} \\
\qquad\quad \text{function } f_j \text{(actual parameter)} \\
\mathcal{N}[\![h(expr_1, \ldots, expr_n)]\!]\ \phi\ \eta\ \rho = \\
\qquad \texttt{mkset}(\ (\mathcal{E}[\![expr_1]\!]\ \phi\ \eta\ \rho), \ldots, (\mathcal{E}[\![expr_n]\!]\ \phi\ \eta\ \rho), \emptyset, l) \\
\qquad\quad \otimes^{\cup} (\mathcal{N}[\![expr_1]\!]\ \phi\ \eta\ \rho) \otimes^{\cup} \ldots \otimes^{\cup} (\mathcal{N}[\![expr_n]\!]\ \phi\ \eta\ \rho)
\end{array}$$

where `condn` is defined as in section 2.7.3 page 28.
and the function `mkset` is defined as follows:

$$\begin{array}{l}
\texttt{mkset}\ (v_1, \ldots, v_n, f, i) = \\
\qquad \texttt{if}\ (v_1 = \perp \vee \ldots \vee v_n = \perp)\ \texttt{then}\ \langle \emptyset, \ldots, \emptyset \rangle \\
\qquad \texttt{else}\ \langle \emptyset, \ldots, \emptyset, \{\langle v_1, \ldots, v_n, f \rangle\}_i, \emptyset, \ldots, \emptyset \rangle
\end{array}$$

and the operator $\otimes^{\cup}$ is defined as in section 2.7.3 page 28.

The function that computes the fix-point is the same as the one for the fisrt order case. It is defined in section 2.7.3 page 29.

3.8 A second order strictness interpreter using mfg

As in the first order case (see section 2.7.4), we define the two-point domain **2** on which we are constructing the strictness interpreter.

$$\mathbf{2} = \{0, 1\}, \qquad 0 \sqsubseteq 1$$

Elements of $\mathcal{D}^*$ are mapped by an abstract function α to an element of **2**. The $\perp$ element is mapped to 0 and other values are mapped to 1. The abstract function is defined as follows:

$$\begin{aligned} &\alpha : \mathcal{D}^* \to \mathbf{2} \\ &\alpha(d) = \texttt{if } d = \perp \texttt{ then 0 else 1} \end{aligned}$$

Variables, parameters and functional environments are redefined by substituting all $\mathcal{D}^*$ by **2**:

$$\begin{aligned} \rho &\in \mathbf{2}^n \\ \eta &\in E \equiv \wp(\mathbf{2}^n \to \mathbf{2}) \\ \phi &\in \Phi \equiv \wp(\mathbf{2}^n \times E \times \mathbf{2})^{p+q} \end{aligned}$$

The translation of the $\mathcal{E}$ function to $\mathcal{E}^\sharp$ is straightforward. The evaluation function of expressions is redefined as follow:

$$\mathcal{E}^\sharp[\![expr]\!] : \Phi \to E \to \mathbf{2}^n \to \mathbf{2}$$

Note that $\mathcal{E}^\sharp$ must verifies safety properties. It means that for all $expr \in \mathbb{E}xpr, \eta \in E, \phi \in \Phi$ and $\rho \in \mathcal{D}^{*n}$, $\mathcal{E}^\sharp$ must verify:

$$\begin{aligned} &\alpha(\mathcal{E}[\![\mathrm{c}]\!]\ \phi\ \rho) \sqsubseteq \mathcal{E}^\sharp[\![\mathrm{c}]\!]\ \phi\ \rho \\ &\alpha(\mathcal{E}[\![x_i]\!]\ \phi\ \rho) \sqsubseteq \mathcal{E}^\sharp[\![x_1]\!]\ \phi\ \rho \\ &\alpha(\mathcal{E}[\![p_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho) \\ &\qquad \sqsubseteq \mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \rho \wedge \ldots \wedge \mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \rho \\ &\alpha(\mathcal{E}[\![\underline{\texttt{if}}(expr_1, expr_2, expr_3)]\!]\ \phi\ \rho \\ &\qquad \sqsubseteq \mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \rho \wedge (\mathcal{E}^\sharp[\![expr_2]\!]\ \phi\ \rho \vee \mathcal{E}^\sharp[\![expr_3]\!]\ \phi\ \rho) \\ &\alpha(\mathcal{E}[\![f_i(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho) \\ &\qquad \sqsubseteq (\phi_j \eta_\perp)(\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \rho \wedge \ldots \wedge \mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \rho) \\ &\alpha(\mathcal{E}[\![g_i(expr_1, \ldots, expr_n) f_j]\!]\ \phi\ \rho) \\ &\qquad \sqsubseteq (\phi_{p+i}(\phi_j \eta_\perp))(\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \rho, \ldots, \mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \rho) \\ &\alpha(\mathcal{E}[\![g_i(expr_1, \ldots, expr_n) h]\!]\ \phi\ \rho) \\ &\qquad \sqsubseteq (\phi_{p+i}(\phi_j \eta))(\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \rho, \ldots, \mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \rho) \\ &\alpha(\mathcal{E}[\![h(expr_1, \ldots, expr_n)]\!]\ \phi\ \rho) \\ &\qquad \sqsubseteq \eta(\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \rho \wedge \ldots \wedge \mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \rho) \end{aligned}$$

In this way, we examine the strictness properties of expressions. Explanations are the same than in the first order case, see section 2.7.4. The

semantics for $\mathcal{E}^\sharp$ follows:

$$\mathcal{E}^\sharp[\![c]\!]\ \phi\ \eta\ \rho = d$$
$$\mathcal{E}^\sharp[\![x_i]\!]\ \phi\ \eta\ \rho = \rho_i$$
$$\mathcal{E}^\sharp[\![p_i(expr_1,\ldots,expr_n)]\!]\ \phi\ \eta\ \rho = (\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \eta\ \rho \wedge \ldots \wedge \mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \eta\ \rho)$$
$$\mathcal{E}^\sharp[\![\underline{\texttt{if}}(expr_1,expr_2,expr_3)]\!]\ \phi\ \eta\ \rho = (\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \eta\ \rho \wedge (\mathcal{E}^\sharp[\![expr_2]\!]\ \phi\ \eta\ \rho \vee \mathcal{E}^\sharp[\![expr_3]\!]\ \phi\ \eta\ \rho))$$
$$\mathcal{E}^\sharp[\![f_i(expr_1,\ldots,expr_n)]\!]\ \phi\ \eta\ \rho = (\bar{\phi}_i\ \bar{\eta_\perp})(\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \eta\ \rho,\ldots,\mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \eta\ \rho)$$
$$\mathcal{E}^\sharp[\![g_i(expr_1,\ldots,expr_n)h]\!]\ \phi\ \eta\ \rho = (\bar{\phi}_{p+i}\ \bar{\eta})(\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \eta\ \rho,\ldots,\mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \eta\ \rho)$$
if h is a parameter name (formal parameter)
$$\mathcal{E}^\sharp[\![g_i(expr_1,\ldots,expr_n)f_j]\!]\ \phi\ \eta\ \rho = (\bar{\phi}_{p+i}(\bar{\phi}_j\ \bar{\eta_\perp}))(\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \eta\ \rho,\ldots,\mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \eta\ \rho)$$
if the parameter value is the named function f_j(actual parameter)
$$\mathcal{E}^\sharp[\![h(expr_1,\ldots,expr_n)]\!]\ \phi\ \eta\ \rho = \bar{\eta}(\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \eta\ \rho,\ldots,\mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \eta\ \rho)$$

The signature of the need function is defined as follows:

$$\mathcal{N}^\sharp[\![expr]\!] : \Phi \to E \to \mathbf{2}^{*n} \to \mathcal{C}$$

The semantics is straightforward with respect to the previous one defined for the concrete case.

$$\mathcal{N}^\sharp[\![c]\!]\ \phi\ \eta\ \rho = \langle\emptyset,\ldots,\emptyset\rangle$$
$$\mathcal{N}^\sharp[\![x_i]\!]\ \phi\ \eta\ \rho = \langle\emptyset,\ldots,\emptyset\rangle$$
$$\mathcal{N}^\sharp[\![p_i(expr_1,\ldots,expr_n)]\!]\ \phi\ \eta\ \rho = (\mathcal{N}^\sharp[\![expr_1]\!]\ \phi\ \eta\ \rho) \otimes^\cup \ldots \otimes^\cup (\mathcal{N}^\sharp[\![expr_n]\!]\ \phi\ \eta\ \rho)$$
$$\mathcal{N}^\sharp[\![\underline{\texttt{if}}(expr_1,expr_2,expr_3)]\!]\ \phi\ \eta\ \rho = \texttt{condn}(expr_1,expr_2,expr_3)$$
$$\mathcal{N}^\sharp[\![f_i(expr_1,\ldots,expr_n)]\!]\ \phi\ \eta\ \rho = \texttt{mkset}(\ (\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \eta\ \rho),\ldots,(\mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \eta\ \rho),\emptyset,i) \otimes^\cup(\mathcal{N}^\sharp[\![expr_1]\!]\ \phi\ \eta\ \rho) \otimes^\cup \ldots \otimes^\cup (\mathcal{N}^\sharp[\![expr_n]\!]\ \phi\ \eta\ \rho)$$
$$\mathcal{N}^\sharp[\![g_i(expr_1,\ldots,expr_n)h]\!]\ \phi\ \eta\ \rho = \texttt{mkset}(\ (\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \eta\ \rho),\ldots,(\mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \eta\ \rho),\eta,i) \otimes^\cup(\mathcal{N}^\sharp[\![expr_1]\!]\ \phi\ \eta\ \rho) \otimes^\cup \ldots \otimes^\cup (\mathcal{N}^\sharp[\![expr_n]\!]\ \phi\ \eta\ \rho)$$
if h is a parameter name (formal parameter)
$$\mathcal{N}^\sharp[\![g_i(expr_1,\ldots,expr_n)f_j]\!]\ \phi\ \eta\ \rho = \texttt{mkset}(\ (\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \eta\ \rho),\ldots,(\mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \eta\ \rho),\phi_j,i) \otimes^\cup(\mathcal{N}^\sharp[\![expr_1]\!]\ \phi\ \eta\ \rho) \otimes^\cup \ldots \otimes^\cup (\mathcal{N}^\sharp[\![expr_n]\!]\ \phi\ \eta\ \rho)$$
if the parameter value is the named function f_j(actual parameter)
$$\mathcal{N}^\sharp[\![h(expr_1,\ldots,expr_n)]\!]\ \phi\ \eta\ \rho = \texttt{mkset}(\ (\mathcal{E}^\sharp[\![expr_1]\!]\ \phi\ \eta\ \rho),\ldots,(\mathcal{E}^\sharp[\![expr_n]\!]\ \phi\ \eta\ \rho),\emptyset,l) \otimes^\cup(\mathcal{N}^\sharp[\![expr_1]\!]\ \phi\ \eta\ \rho) \otimes^\cup \ldots \otimes^\cup (\mathcal{N}^\sharp[\![expr_n]\!]\ \phi\ \eta\ \rho)$$

This algorithm detects whether a function is strict in its arguments or not. Since the abstract semantics is independent of parameter passing, we are now able to detect cases where call-by-name can be transformed into call-by-value. The difficulties in the second order case was situated in the functional argument.

3.9 Examples

Let us examine the example described in section 3.6 page 50. If we call `map 0 fibfact`, the result of the evaluation will be 0. Hence, we know that `map` function will be strict in its first argument and we can replace its call-by-name parameter passing by call-by-value. But what about the function `fibfact` ? `fibfact` is the functional parameter, there is no interest to replace a function by 0 or 1. In the semantics that we have proposed above, the minimal function graph of this argument is computed. Then, it is passed to the second order function in order to evaluate the result of the inital function call.

Chapter 4

Conclusion

After having introduced the mathematical tools for our development, we have defined a first order functional language. First, we have presented how to write the semantics of a *strict* language and difficulties we have encountered to define a *non-strict* language starting from the former. For these first semantics we have worked on a simple set of values $\mathcal{D}$ and we were interested in the integration of lists. We have seen that fix-point computation for these semantics does not change no matter how complex the set of value or the parameter passing mode are. Nevertheless, we have seen that the *strict* evaluation is more efficient than the *non-strict* evaluation but at the cost of not terminating in many cases. In order to improve the lazy evaluation, we have then studied cases where call-by-name could be replaced by call-by-value. To do this, we have studied another fix-point computation which is based on *minimal function graph*. We have then applied strictness analysis on our language with a two points domains on which all well-defined values are mapped to 1 and undefined are mapped to 0. This algorithm allowed us to detect cases where call-by-name can be transformed into call-by-value.

The next step was to study higher order cases. We have restricted our study to the second order case and gone deeper into the point tackled in the first chapter.

It is certain that our developments are incomplete. Even if the integration of list is a good idea, the strictness analysis applied to languages supporting lists could be better developped. Instead of working on a two points domain, we may apply strictness analysis on a four points domain which is a non-flat domain as described in [Wad87].

A weak point of our approach is that our second order language does not support curried functions. A next step should be to take currification into account. We have not presented in this document an interpreter which performs strictness analysis at compile-time. It would be interesting to implement such an interpreter and compare its computation time with the one of a call-by-name interpreter. Such an interpreter is described for the

fisrt order case in [Pol96]. The author shows how much computation time of function calls is reduced using this technique but she does not generalize in all cases. Another further work should be to study the efficiency of other fix-point algorithms in terms of space- and time-consumption. In [LCVH93], the authors present a general fix-point algorithm for abstract interpretation. The functioning mode of this algorithm is similar to the one of minimal function graph but is applicable to many other abstract interpretation areas. We could also extend our work directly to the higher order case as explain in [BHA86] [DJR97].

Bibliography

[BHA86] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness Analysis For Higher-Order Functions. *Science of Computer Programming*, 7:249–278, 1986.

[DJM86] Neil D. Jones and Alan Mycroft. Data Flow Analysis of Applicative Programs Using Minimal Function Graphs. In *13th Symp. on Principles of Prog. Languages*, pages 296–306, St. Petersbug, Florida, January 1986.

[DJR97] Neil D. Jones and Mads Rosendahl. Higher Order Minimal Function Graph. Technical report, University of Copenhagen, University of Roskilde, 1997.

[HPF99] Paul Hudak, John Peterson, and Joseph H. Fasel. *A Gentle Introduction to Haskell 98*, 1999.

[LC] Baudouin Le Charlier. *Interprétation Abstraite.* Course notes.

[LC92] B. Le Charlier. L'Analyse Statique des Programmes par Interprétation Abstraite. *Nouvelles de la Science et des Technologies*, 9(4):19–25, 1992.

[LCVH93] B. Le Charlier and P. Van Hentenryck. A general top-down fixpoint algorithm (revised version). Technical Report 93-22, Institute of Computer Science, University of Namur, Belgium, (also Brown University), jun 1993.

[Ler77] H. Leroy. *La Fiabilité des Programmes*, chapter 5 - Sémantique Mathématique, pages 5.1–5.15. Ecole d'été de l'A.F.C.E.T., 1977.

[Mau95] Michel Mauny. *Functional Programming Using Caml Light*, 1995.

[Myc80] Alan Mycroft. The Theory and Pratice of Transforming Call-by-Need into Call-by-Value. In *International Symposium on Programming*, volume 83 of LNCS, pages 269–281, Paris, France, April 1980. Springer-Verlag.

[Pol96] Isabelle Pollet. Analyse de Strictness de Langages Applicatifs par l' Interprétation Abstraite Master's thesis, University of Namur, 1996-97

[Ros95] Mads Rosendahl. *Introduction to Abstract Interpretation*, 1995.

[Ros01] Mads Rosendahl. *Introduction to Domain Theory*. DIKU, Computer Science University of Copenhagen, 2001.

[Rou97] Jacques Rouablé. *Programmation en Caml*. Eyrolles, 1997.

[Sto77] J. Stoy. *Denotational Semantics*, chapter 5 - The lambda-calculus, pages 52–77. M.I.T. Press, 1977.

[Ten81] R.D. Tennet. *Principles of Programming Languages*, chapter 13 - Formal Semantics, pages 211–249. Prentice-Hall, 1981.

[Wad87] Phil Wadler. Strictness Analysis on Non-Flat Domains (by Abstract Interpretation Over Finite Domains. In *Absract Interpretation of Declarative Languages*, chapter 12, pages 266–275. Ellis-Horwood edition, 1987.

Appendix A

First order case implementations

A.1 Call-by-value interpreter

```
exception NotAValidType of string;;
exception EmptyList of string;;
exception EmptyString of string;;
exception LengthError of string;;
exception NotAPair of string;;
exception IfError of string;;

(* Basic values *)

type val = I of int
         | S of string ;;

(* Expressions *)

type expr = INT of int
          | STR of string
          | VAR of string
          | CALL of string*(expr list)
          | IF of expr*expr*expr
          | SUCC of expr
          | PRED of expr
          | CAT of expr*expr
          | EQU of expr*expr ;;

(* Program *)

type prog = PROG of ((string*(string list)*expr) list) ;;

(* Variable environment *)

type rho = RHO of (string*val) list;;
```

```
(* Function environment *)

type phi = PHI of (string -> ((val list) -> val)) ;;

(* Some useful functions *)

let rec valofs s = fun []       -> raise (EmptyList "in valofs function")
                 |   ((a,b)::l) -> if (s=a) then b else valofs s l ;;

let rec zip = fun [] [] -> []
            |    (a::r) (b::s) -> (a,b)::(zip r s)
            |     _  _  -> raise (LengthError "in zip function");;

let istrue = fun (I 0) -> false
           |     (I _) -> true
           |       _   -> false;;

(* Primitives - buit-in functions *)

let succ = fun (I n) -> (I (n+1))
         |       _   -> raise (NotAValidType "in succ function");;

let pred = fun (I n) -> (I (n-1))
         |       _   -> raise (NotAValidType "in pred function");;

let equ = fun (I i , I j) -> if (i=j) then (I 1) else (I 0)
        |     (S s , S t) -> if (s=t) then (I 1) else (I 0)
        |     ( _  ,  _ ) -> raise (NotAValidType "in equ function");;

let cat = fun (S s , S t) -> S (s^t)
        |     ( _ , _ )  -> raise (NotAValidType "in cat function");;

(* Expression evaluation *)

let rec e (RHO rho) (PHI phi) =
  fun  (INT i)      -> (I i)
   |   (STR s)      -> (S s)
   |   (VAR s)      -> valofs s rho
   |   (CALL (f,l)) ->
         (phi f)  (map (e (RHO rho) (PHI phi)) l)
   |   (IF (c, a, b)) ->
     if (istrue (e (RHO rho) (PHI phi) c))
     then (e (RHO rho) (PHI phi) a)
     else (e (RHO rho) (PHI phi) b)
   |   (SUCC x)     -> succ (e (RHO rho) (PHI phi) x)
   |   (PRED x)     -> pred (e (RHO rho) (PHI phi) x)
   |   (CAT (x,y))  ->
         cat ( (e (RHO rho) (PHI phi) x),
           (e (RHO rho) (PHI phi) y)  )
   |   (EQU (x,y))  ->
         equ ( (e (RHO rho) (PHI phi) x),
           (e (RHO rho) (PHI phi) y) );;

(* Program evaluation *)
```

```
let rec p = fun (PROG []) (PHI phi) f vl -> raise (EmptyList "in prog")
          |     (PROG ((fn, pl, expr)::r)) (PHI phi) f vl ->
                  if (f=fn)
                  then (e (RHO (zip pl vl)) (PHI phi) expr)
                  else p (PROG r) (PHI phi) f vl;;

let rec fixphi (PROG prog) f vl =
        p (PROG prog) (PHI (fixphi (PROG prog))) f vl ;;
```

A.2 Call-by-name interpreter

```
(* Compare this implementation with the call-by-value case *)
(* Basic values have not changed *)
(* Value may be not evaluated immediately *)

type nval ==  (unit -> val) ;;

(* Variable environment *)

type rho = RHO of (string * nval) list;;

(* Function environment *)

type phi = PHI of (string -> (nval list) -> nval) ;;

(* The trick - Reduction function *)

let myin  a = a() ;;

(* Expression evaluation. Caml use a strict evaluation. This expression
   evaluation simulates a call-by-name interpreter *)

let rec en (RHO rho) (PHI phi) =
  fun    (INT i)  ()  -> (I i)
   |     (STR s)  ()  -> (S s)
   |     (VAR s)  ()  ->  myin (valofs s rho)
   |     (CALL (f,l)) () ->
           myin ((phi f) ((map (en (RHO rho) (PHI phi)) l )))
   |     (IF (c, a, b)) () ->
            if istrue (myin (en (RHO rho) (PHI phi) c))
            then (en (RHO rho) (PHI phi) a ())
            else (en (RHO rho) (PHI phi) b ())
   |     (SUCC x) () -> (succ (myin (en (RHO rho) (PHI phi) x)))
   |     (PRED x) () -> (pred  (myin (en (RHO rho) (PHI phi) x)))
   |     (CAT (x,y)) () ->
           ( cat ( myin (en (RHO rho) (PHI phi) x),
                   myin (en (RHO rho) (PHI phi) y)  ) )
   |     (EQU (x,y))  () ->
           ( equ ( (myin (en (RHO rho) (PHI phi) x)),
                   (myin (en (RHO rho) (PHI phi) y)) ) ) ;;
```

A.3 Call-by-value interpreter with lists

```
(* Compare this implementation with the call-by-value case without list *)
(* Basic values: D = D U { NULL } and D* = D U (D* x D* ) *)

type val = I of int
         | S of string
         | P of val*val
         | NULL ;;

(* In expressions, we add the constructor CONS and destructor CAR,CDR *)

type expr = NIL
          (* ... *)
          | CONS of expr*expr
          | CAR of expr
          | CDR of expr
          | ATOM of expr ;;

(* The three add-on primitives *)

let cons = fun x  y  -> P (x , y);;

let car = fun (P (x,y)) -> x
        |          _     -> raise (NotAPair "in car function");;

let cdr = fun (P (x,y)) -> y
        |          _     -> raise (NotAPair "in cdr function");;

(* We have also add a new primitive to test if an element is an atom or not *)

let atom = fun (I x) -> (I 1)
         |     (S x) -> (I 1)
         |     NULL  -> (I 1)
         |     (P (x,y)) -> (I 0);;

(* Expression evaluation *)

let rec e (RHO rho) (PHI phi) =
  fun NIL          -> NULL
   (*  ...  *)
   |  (CONS (x,y)) ->
        cons  (e (RHO rho) (PHI phi) x)
              (e (RHO rho) (PHI phi) y)
   |  (CAR x) -> car (e (RHO rho) (PHI phi) x)
   |  (CDR x) -> cdr (e (RHO rho) (PHI phi) x)
   |  (ATOM x) -> atom (e (RHO rho) (PHI phi) x);;
```

A.4 Call-by-name interpreter with lazy lists

```
(* Compare this implementation with respect to call-by-name without lists *)

type expr = NIL
```

```
          (* ... *)
          | CONS of expr*expr
          | CAR of expr
          | CDR of expr
          | ATOM of expr ;;

(* Tricks to implement lazy lists *)

let myout a () = a ;; (* Abstraction function *)
let myin  a = a() ;;  (* Reduction function *)

(* The three add-on primitives *)

let cons = fun x  y  -> myout (P (x , y));;

let car x = match myin x
            with (P(x,y))  -> x
             |      _      -> raise (NotAPair "in car function");;

let cdr x = match myin x
            with  (P(x,y)) -> y
             |       _      -> raise (NotAPair "in cdr function");;

let atom = fun (I x) -> (I 1)
         |     (S x) -> (I 1)
         |     NULL  -> (I 1)
         |     (P (x,y)) -> (I 0);;

(* Expression evaluation *)

let rec en (RHO rho) (PHI phi) =
   fun NIL       () -> NULL
       (* ... *)
 |     (CONS (x,y)) () ->
           myin ( cons ((en (RHO rho) (PHI phi) x ))
                  ((en (RHO rho) (PHI phi) y)) )
 |     (CAR x) () -> myin (car ( (en (RHO rho) (PHI phi) x)))
 |     (CDR x) () -> myin (cdr ( (en (RHO rho) (PHI phi) x)))
 |     (ATOM x) () -> (atom (myin (en (RHO rho) (PHI phi) x)));;
```

A.5 A mfg implementation

```
(* We add a special value BOT in the set of values *)

type bval = I of int
          | S of string
          | P of bval*bval
          | NULL
          | BOT ;;

(* The function environment is defined as a tuple *)

type phi = PHI of (string * (bval list) * bval ) list ;;
```

```
(* Primitives are redefined such that they support the BOT value *)

let succ = fun (I n) -> I (n+1)
         |     BOT   -> BOT
         |     _     -> raise (NotAValidType "in succ function");;

let pred = fun (I n) -> I (n-1)
         |      BOT  -> BOT
         |       _   -> raise (NotAValidType "in pred function");;

let equ = fun      ( BOT ,  _  ) -> BOT
           |       (  _  , BOT ) -> BOT
           |       (I i  ,  I j) -> if (i=j) then I 1 else I 0
           |       (S s  ,  S t) -> if (s=t) then I 1 else I 0
           |       ( NULL, NULL) -> I 1
           |       (  _  ,  NULL ) -> I 0
           |       ( NULL , _ ) -> I 0
           |       ( P(x,y) , _ ) -> I 0
           |       (  _  , P(a,b)) -> I 0
           |       (  _  ,  _ ) -> raise (NotAValidType "in equ function");;

let cat = fun ( BOT , _ ) -> BOT
        |     ( _ , BOT ) -> BOT
        |     (S s , S t) -> S (s^t)
        |     ( _ , _ )  -> raise (NotAValidType "in cat function");;

let cons = fun BOT   _  -> BOT
         |      _   BOT -> BOT
         |      x    y  -> P (x , y);;

let car = fun BOT -> BOT
        |     (P (x,y)) -> x
        |         _     -> raise (NotAPair "in car function");;

let cdr = fun  BOT -> BOT
        |     (P (x,y)) -> y
        |         _     -> raise (NotAPair "in cdr function");;

let atom = fun (I x) -> I 1
         |     (S x) -> I 1
         |     (NULL)  -> I 1
         |     (P (x,y)) -> I 0
         |     BOT  -> I 0;;

(* Some useful functions *)

let rec seek f = fun [] -> []
               |     ((e, l)::r) -> if (f=e) then l else seek f r;;

let rec iseq = fun (BOT) (BOT) -> true
             |     (I x) (I y) -> x=y
             |     (S t) (S r) -> t=r
             |     (P(a,b)) (P(x,y)) -> (iseq a x) && (iseq b y)
```

```
            |       _     _   -> false;;

(* Expression evaluation function *)
(* Expression are evaluated as usual except for function call because phi is
   defined as a tuple instead of function *)

let rec e (RHO rho) (PHI phi) =
 fun (* ... *)
  |  (CALL (f,l)) ->
     lookupphi f (map (e (RHO rho) (PHI phi)) l) phi

(* checkargs verify if two lists are equal *)

and checkargs = fun []       []    -> true
              |  (e1::l1) (e2::l2) -> (iseq e1 e2) && (checkargs l1 l2)
              |      _        _    -> false

(* member checks if an element is in a list or not *)

and member x = fun []   -> false
             |   (e::l) -> if (x=e) then true else member x l

(* lookupphi checks if one argument of the arg list is BOT. If it is the case,
   it returns BOT, otherwise it extracts the result value of f from phi *)

and lookupphi f arg phi = if (member (BOT) arg) then BOT
                          else lookupphi1 f arg phi

and lookupphi1 f arg = fun [] -> BOT
                     |  ((f1, arg1, res)::r) ->
                         if (checkargs arg arg1) && (f=f1)
                         then res
                         else lookupphi1 f arg r;;

(* Need: collects the function call that are needed *)

let rec need (RHO rho) (PHI phi) =
 fun NIL      -> []
 |   (INT i)  -> []
 |   (STR s)  -> []
 |   (VAR x)  -> []
 |   (FUNC (f,l)) ->
addneed ( f , (map (e (RHO rho) (PHI phi)) l) , (needlist (RHO rho) (PHI phi) l))
 |   (IF (c,a,b)) ->
      begin
      match (e (RHO rho) (PHI phi) c)
      with (I 0) -> (need (RHO rho) (PHI phi) c)@(need (RHO rho) (PHI phi) b)
      |    (I _) -> (need (RHO rho) (PHI phi) c)@(need (RHO rho) (PHI phi) a)
      |      _   -> (need (RHO rho) (PHI phi) c)
      end
 |   (SUCC x) -> (need (RHO rho) (PHI phi) x)
 |   (PRED x) -> (need (RHO rho) (PHI phi) x)
 |   (CAT (x,y)) -> (need (RHO rho) (PHI phi) y)@(need (RHO rho) (PHI phi) x)
 |   (EQU (x,y)) -> (need (RHO rho) (PHI phi) y)@(need (RHO rho) (PHI phi) x)
```

```
 |   (CONS(x,y)) -> (need (RHO rho) (PHI phi) y)@(need (RHO rho) (PHI phi) x)
 |   (CAR x) -> (need (RHO rho) (PHI phi) x)
 |   (CDR x) -> (need (RHO rho) (PHI phi) x)
 |   (ATOM x) -> (need (RHO rho) (PHI phi) x)

(* needlist computes need for a list of expressions *)

and needlist (RHO rho) (PHI phi) = fun []   -> []
                                 |   (e::l) -> (need (RHO rho) (PHI phi) e)
                                       @(needlist (RHO rho) (PHI phi) l)

(* For function call, if one of the arg is BOT then don't add the closure
   to C *)

and addneed (f, arg, nlist) = if (member BOT arg) then nlist
                              else (f,arg)::nlist;;

(* Some useful functions *)

(* Test the equality of two elements from phi *)

let iseq_phi (f1, l1, v1) (f2, l2, v2) =
              (f1 = f2) && (iseq v1 v2) && (checkargs l1 l2);;

(* Test if an element is in phi or not *)

let rec is_in_phi_list v = fun   []   -> false
                         |     (e::l) -> (iseq_phi v e)
                                         or (is_in_phi_list v l);;

(* Is a set included in another one ? *)

let rec included = fun [] [] -> true
                 | (e::l) m -> (is_in_phi_list e m) && (included l m)
                 |   []   _ -> true;;

(* Test if two phi sets are equal *)

let testphi l1 l2 = (included l1 l2) && (included l2 l1);;

(* Test the equality of two elements from C *)

let iseq_c (f1, l1) (f2, l2) = (f1 = f2) && (checkargs l1 l2);;

(* Test if an element from C is in C or not *)

let rec is_in_c_list v = fun [] -> false
                   |    (e::l)  -> (iseq_c v e) or (is_in_c_list v l);;

(* Is a set included in another one ? *)

let rec included_c = fun [] []   -> true
                   |    (e::l) m -> (is_in_c_list e m) && (included_c l m)
                   |     []  _   -> true;;
```

```
(* Test the equality of two C sets *)

let test_c l1 l2 = (included_c l1 l2) && (included_c l2 l1);;

(* Function that computes fix-point *)
(* phi and C are improved until they are equal *)

let rec iterate phi c prog =
    let phi1 = newphi c phi prog and
        c1   = union (newc c phi prog) c
    in if ((testphi phi phi1) && (test_c c c1)) then (phi1, c1)
       else iterate phi1 c1 prog

(* newphi computes a new phi *)
(* if C is the empty set, then returns the empty set
   otherwise, for each closure in C, we construct a phi *)

and newphi = fun    []      phi prog -> []
           |   ((f,arg)::r) phi prog ->
           (f, arg, e (RHO (buildrho f prog arg)) (PHI phi) (findexp f prog))::
           (newphi r phi prog)

(* newc computes a new C *)
(* if C is the empty set, then returns the empty set
   otherwise, for each closure in C, we collect new closures *)

and newc = fun [] phi prog -> []
         | ((f,arg)::r) phi prog ->
           (need (RHO (buildrho f prog arg)) (PHI phi) (findexp f prog))@
           (newc r phi prog)

(* buildrho constructs a rho for a function f in a program prog with a the list
   of value arg *)

and buildrho = fun f [] arg -> raise (EmptyList "in buildrho")
             |     f ((f1, lvar, exp)::r) arg ->
                     if (f=f1) then (zip lvar arg)
                     else buildrho f r arg

(* findexp returns the expr associated to a function f from a program prog *)

and findexp = fun f [] -> raise (EmptyList "in findexp")
             |     f ((f1, lvar, exp)::r) ->
                     if (f=f1) then exp
                     else findexp f r;;

(* start is a userfriendly way of calling iterate *)

let start f arg (PROG prg) = iterate [] [(f,arg)] prg ;;
```

A.6 A strictness interpreter using mfg

```
(* Compare this case with the interpreter using minimal function graph *)
(* New domain for a strictness interpreter
   We work with the two point domain 2 which contains ZERO and ONE *)

type bval = ZERO | ONE;;

(* Primitives for the strictness interpreter are only the logical 'and' and
   'or' *)

let myand = fun ONE ONE -> ONE
          |      _   _  -> ZERO;;

let myor = fun ZERO ZERO -> ZERO
         |      _    _   -> ONE;;

(* Some useful functions *)

let rec valofs s = fun []       -> raise (EmptyList "in valofs function")
                 |   ((a,b)::l) -> if (s=a) then b else valofs s l ;;

let rec zip = fun [] [] -> []
            |    (a::r) (b::s) -> (a,b)::(zip r s)
            |     _  _  -> raise (LengthError "in zip function");;

let rec iseq = fun (ONE) (ONE) -> true
             |     (ZERO) (ZERO) -> true
             |       _     _   -> false;;

(* E# *)

let rec esharp (RHO rho) (PHI phi) =
 fun NIL          -> ONE
  |  (INT i)      -> ONE
  |  (STR s)      -> ONE
  |  (VAR s)      -> (valofs s rho)
  |  (CALL (f,l)) ->
     lookupphi f (map (esharp (RHO rho) (PHI phi)) l) phi
  |  (IF (c, a, b)) ->  myand (esharp (RHO rho) (PHI phi) c)
                              (myor (esharp (RHO rho) (PHI phi) a)
                                    (esharp (RHO rho) (PHI phi) b))
  |  (SUCC x)     -> (esharp (RHO rho) (PHI phi) x)
  |  (PRED x)     -> (esharp (RHO rho) (PHI phi) x)
  |  (CAT (x,y))  ->
       myand (esharp (RHO rho) (PHI phi) x)
             (esharp (RHO rho) (PHI phi) y)
  |  (EQU (x,y))  ->
       myand (esharp (RHO rho) (PHI phi) x)
     (esharp (RHO rho) (PHI phi) y)
  |  (CONS (x,y)) ->
       myand  (esharp (RHO rho) (PHI phi) x)
      (esharp (RHO rho) (PHI phi) y)
  |  (CAR x) -> (esharp (RHO rho) (PHI phi) x)
```

```
  |  (CDR x) -> (esharp (RHO rho) (PHI phi) x)
  |  (ATOM x) -> (esharp (RHO rho) (PHI phi) x)

and checkargs = fun []       []    -> true
              |  (e1::l1) (e2::l2) -> (iseq e1 e2) && (checkargs l1 l2)
              |      _        _    -> false

and member x = fun []   -> false
             |   (e::l) -> if (x=e) then true else member x l

and lookupphi f arg phi = if (member (ZERO) arg) then ZERO
                          else lookupphi1 f arg phi

and lookupphi1 f arg = fun [] -> ZERO
                     |  ((f1, arg1, res)::r) ->
                        if (checkargs arg arg1) && (f=f1)
                        then res
                        else lookupphi1 f arg r;;

(* N# *)

let rec needsharp (RHO rho) (PHI phi) =
 fun NIL       -> []
 |   (INT i)   -> []
 |   (STR s)   -> []
 |   (VAR x)   -> []
 |   (CALL (f,l)) -> addneed ( f,
                               (map (esharp (RHO rho) (PHI phi)) l),
                               (needlist (RHO rho) (PHI phi) l) )
 |   (IF (c,a,b)) -> (needsharp (RHO rho) (PHI phi) c)@
                     (needsharp (RHO rho) (PHI phi) a)@
                     (needsharp (RHO rho) (PHI phi) b)
 |   (SUCC x)     -> (needsharp (RHO rho) (PHI phi) x)
 |   (PRED x)     -> (needsharp (RHO rho) (PHI phi) x)
 |   (CAT (x,y))  -> (needsharp (RHO rho) (PHI phi) y)@
                     (needsharp (RHO rho) (PHI phi) x)
 |   (EQU (x,y))  -> (needsharp (RHO rho) (PHI phi) y)@
                     (needsharp (RHO rho) (PHI phi) x)
 |   (CONS(x,y))  -> (needsharp (RHO rho) (PHI phi) y)@
                     (needsharp (RHO rho) (PHI phi) x)
 |   (CAR x)      -> (needsharp (RHO rho) (PHI phi) x)
 |   (CDR x)      -> (needsharp (RHO rho) (PHI phi) x)
 |   (ATOM x)     -> (needsharp (RHO rho) (PHI phi) x)

and needlist (RHO rho) (PHI phi) = fun []   -> []
                                 |   (e::l) -> (needsharp (RHO rho) (PHI phi) e)
                                        @(needlist (RHO rho) (PHI phi) l)

and addneed (f, arg, nlist) = if (member ZERO arg) then nlist
                              else (f,arg)::nlist;;

(* Fixpoint computation does not change with respect to the mfg
   implementation *)
```

Appendix B

Second order case implementations

B.1 Call-by-value interpreter

```
(* Basic Values *)

type val = I of int
         | S of string ;;

(* Expressions *)

type expr = INT of int
          | STR of string
          | VAR of string
          | CALL of string*(expr list)*string
          | IF of expr*expr*expr
          | SUCC of expr
          | PRED of expr
          | CAT of expr*expr
          | EQU of expr*expr ;;

(* Variable environment *)

type rho = RHO of (string*val) list;;

(* Function environment *)

type phi = PHI of (string -> (val list) -> string -> val) ;;

(* Program *)

type prog = PROG of ((string*(string list)*string*expr) list) ;;

(* Some useful functions *)

let rec valofs s = fun []       -> raise (EmptyList "in valofs function")
```

```
                |   ((a,b)::l) -> if (s=a) then b else valofs s l ;;

let rec zip = fun [] [] -> []
            |    (a::r) (b::s) -> (a,b)::(zip r s)
            |     _  _  -> raise (LengthError "in zip function");;

let istrue = fun (I 0) -> false
           |     (I _) -> true
           |       _   -> false;;

(* Primitives *)

let succ = fun (I n) -> (I (n+1))
 |       _   -> raise (NotAValidType "in succ function");;

let pred = fun (I n) -> (I (n-1))
 |       _   -> raise (NotAValidType "in pred function");;

let equ = fun (I i , I j) -> if (i=j) then (I 1) else (I 0)
|     (S s , S t) -> if (s=t) then (I 1) else (I 0)
|     ( _  ,  _ ) -> raise (NotAValidType "in equ function");;

let cat = fun (S s , S t) -> S (s^t)
|     ( _ , _ )  -> raise (NotAValidType "in cat function");;


(* Expression evaluation function *)

let lookupf s pn pv = if (s=pn) then pv else s;;

let rec e (RHO rho) (PHI phi) pn pv =
  fun  (INT i)        -> (I i)
   |   (STR s)        -> (S s)
   |   (VAR s)        -> valofs s rho
   |   (CALL (f,l,s)) ->
         if (f = pn)
         then (phi pv) (map (e (RHO rho) (PHI phi) pn pv) l) (lookupf s pn pv)
         else (phi f) (map (e (RHO rho) (PHI phi) pn pv) l)  (lookupf s pn pv)
   |   (IF (c, a, b)) ->
     if (istrue (e (RHO rho) (PHI phi) pn pv c))
     then (e (RHO rho) (PHI phi) pn pv a)
     else (e (RHO rho) (PHI phi) pn pv b)
   |   (SUCC x)     -> succ (e (RHO rho) (PHI phi) pn pv x)
   |   (PRED x)     -> pred (e (RHO rho) (PHI phi) pn pv x)
   |   (CAT (x,y))  ->
         cat ( (e (RHO rho) (PHI phi) pn pv x),
           (e (RHO rho) (PHI phi) pn pv y)  )
   |   (EQU (x,y))  ->
         equ ( (e (RHO rho) (PHI phi) pn pv x),
           (e (RHO rho) (PHI phi) pn pv y) );;

(* Program evaluation *)

let rec p = fun (PROG []) (PHI phi) f vl pv -> raise (EmptyList "in prog")
```

```
          |     (PROG ((fn, pl, q, expr)::r)) (PHI phi) f vl pv ->
                  if (f=fn)
                  then (e (RHO (zip pl vl)) (PHI phi) q pv expr)
                  else p (PROG r) (PHI phi) f vl pv;;

let rec fixphi (PROG prog) f vl =
        p (PROG prog) (PHI (fixphi (PROG prog))) f vl;;
```

B.2 Call-by-name interpreter

```
(* Compare this implementation with the second order call-by-value interpreter *)
(* Basic values have not changed *)
(* Value may be not evaluated immediately *)

type val = I of int
         | S of string;;

type nval ==  (unit -> val) ;;

(* Variable environement *)

type rho = RHO of (string * nval) list;;

(* Function environment *)

type phi = PHI of (string -> (nval list) -> string -> nval) ;;

(* The trick - Reduction function *)

let myin  a = a() ;;

(* Expression evaluation *)

let lookupf s pn pv = if (s=pn) then pv else s;;

let rec en (RHO rho) (PHI phi) pn pv =
  fun    (INT i)  ()  -> (I i)
   |     (STR s)  ()  -> (S s)
   |     (VAR s)  ()  ->  myin (valofs s rho)
   |     (CALL (f,l,s)) () ->
  if (f=pn)
  then
    myin ((phi pv) ((map (en (RHO rho) (PHI phi) pn pv) l )) (lookupf s pn pv))
  else
    myin ((phi f) ((map (en (RHO rho) (PHI phi) pn pv) l)) (lookupf s pn pv))
   |     (IF (c, a, b)) () ->
           if istrue (myin (en (RHO rho) (PHI phi) pn pv c))
           then (en (RHO rho) (PHI phi) pn pv a ())
           else (en (RHO rho) (PHI phi) pn pv b ())
   |     (SUCC x) () -> (succ (myin (en (RHO rho) (PHI phi) pn pv x)))
   |     (PRED x) () -> (pred  (myin (en (RHO rho) (PHI phi) pn pv x)))
   |     (CAT (x,y)) () ->
           ( cat ( myin (en (RHO rho) (PHI phi) pn pv x),
```

```
                  myin (en (RHO rho) (PHI phi) pn pv y)  ) )
   |     (EQU (x,y))  () ->
           ( equ ( (myin (en (RHO rho) (PHI phi) pn pv x)),
                   (myin (en (RHO rho) (PHI phi) pn pv y)) ) ) ;;
```

B.3 Call-by-value interpreter with lists

```
(* Compare this implementation with the second order call-by-value interpreter
   without lists *)
(* Basic values: D = D U { NULL } and D* = D U (D* x D* ) *)

type val = I of int
         | S of string
         | P of val*val
         | NULL ;;

(* In expressions, we add the constructor CONS and destructor CAR,CDR *)

type expr = NIL
          (* ... *)
          | CONS of expr*expr
          | CAR of expr
          | CDR of expr
          | ATOM of expr ;;

(* The three add-on primitives *)

let cons = fun x  y  -> P (x , y);;

let car = fun (P (x,y)) -> x
        |          _     -> raise (NotAPair "in car function");;

let cdr = fun (P (x,y)) -> y
        |          _     -> raise (NotAPair "in cdr function");;

(* We have also add a new primitive to test if an element is an atom or not *)

let atom = fun (I x) -> (I 1)
         |     (S x) -> (I 1)
         |     NULL  -> (I 1)
         |     (P (x,y)) -> (I 0);;


(* Expression evaluation *)

let lookupf s pn pv = if (s=pn) then pv else s;;

let rec e (RHO rho) (PHY phy) pn pv =
  fun NIL          -> NULL
   (* ... *)
   |  (CONS (x,y)) ->
        cons  (e (RHO rho) (PHY phy) pn pv x)
              (e (RHO rho) (PHY phy) pn pv y)
```

```
   |  (CAR x) -> car (e (RHO rho) (PHY phy) pn pv x)
   |  (CDR x) -> cdr (e (RHO rho) (PHY phy) pn pv x)
   |  (ATOM x) -> atom (e (RHO rho) (PHY phy) pn pv x);;
```

B.4 Call-by-name interpreter with lazy lists

```
(* Compare this implemenation with respect to call-by-name without lists *)

type expr = NIL
          (* ... *)
          | CONS of expr*expr
          | CAR of expr
          | CDR of expr
          | ATOM of expr ;;

(* Tricks to implement lazy lists *)

let myout a () = a ;; (* Abstraction function *)
let myin  a = a() ;;  (* Reduction function *)

(* The three add-on primitives *)

let cons = fun x  y  -> myout (P (x , y));;

let car x = match myin x
            with (P(x,y))  -> x
             |      _      -> raise (NotAPair "in car function");;

let cdr x = match myin x
            with  (P(x,y)) -> y
             |       _      -> raise (NotAPair "in cdr function");;

let atom = fun (I x) -> (I 1)
         |     (S x) -> (I 1)
         |     NULL  -> (I 1)
         |     (P (x,y)) -> (I 0);;

(* Expression evaluation *)

let lookupf s pn pv = if (s=pn) then pv else s;;

let rec en (RHO rho) (PHY phy) pn pv =
   fun NIL       () -> NULL
       (* ... *)
 |     (CONS (x,y)) () ->
           myin ( cons ((en (RHO rho) (PHY phy) pn pv x ))
                       ((en (RHO rho) (PHY phy) pn pv y)) )
 |     (CAR x) () -> myin (car ( (en (RHO rho) (PHY phy) pn pv x)))
 |     (CDR x) () -> myin (cdr ( (en (RHO rho) (PHY phy) pn pv x)))
 |     (ATOM x) () -> (atom (myin (en (RHO rho) (PHY phy) pn pv x)));;
```

B.5 A mfg implementation

```
(* We add a special value BOT in the set of values *)

type bval = I of int
          | S of string
          | P of bval*bval
          | NULL
          | BOT ;;

(* The function environment is defined as a tuple *)

type phi = PHI of (string * (bval list) * bval ) list ;;

(* Primitives are redefined such that they support the BOT value *)

let succ = fun (I n) -> I (n+1)
         |     BOT   -> BOT
         |     _     -> raise (NotAValidType "in succ function");;

let pred = fun (I n) -> I (n-1)
         |      BOT  -> BOT
         |       _   -> raise (NotAValidType "in pred function");;

let equ = fun     ( BOT ,  _  ) -> BOT
            |     (  _  , BOT ) -> BOT
            |     (I i  ,  I j) -> if (i=j) then I 1 else I 0
            |     (S s  ,  S t) -> if (s=t) then I 1 else I 0
            |     ( NULL, NULL) -> I 1
            |     ( _ ,  NULL ) -> I 0
            |     ( NULL , _ ) -> I 0
            |     ( P(x,y) , _ ) -> I 0
            |     ( _  , P(a,b)) -> I 0
            |     ( _  ,  _ ) -> raise (NotAValidType "in equ function");;

let cat = fun ( BOT , _ ) -> BOT
        |     ( _ , BOT ) -> BOT
        |     (S s , S t) -> S (s^t)
        |     ( _ , _ )  -> raise (NotAValidType "in cat function");;

let cons = fun BOT   _  -> BOT
         |      _   BOT -> BOT
         |      x    y  -> P (x , y);;

let car = fun BOT -> BOT
        |     (P (x,y)) -> x
        |         _     -> raise (NotAPair "in car function");;

let cdr = fun  BOT -> BOT
        |     (P (x,y)) -> y
        |         _     -> raise (NotAPair "in cdr function");;

let atom = fun (I x) -> I 1
         |     (S x) -> I 1
```

```
        |     (NULL)  -> I 1
        |     (P (x,y)) -> I 0
        |     BOT  -> I 0;;

(* Some useful functions *)

let rec seek f = fun [] -> []
               |     ((e, l)::r) -> if (f=e) then l else seek f r;;

let rec iseq = fun (BOT) (BOT) -> true
             |     (I x) (I y) -> x=y
             |     (S t) (S r) -> t=r
             |     (P(a,b)) (P(x,y)) -> (iseq a x) && (iseq b y)
             |       _     _   -> false;;

(* Expression evaluation function *)
(* Expression are evaluated as usual except for function call because phi is
   defined as a tuple instead of function *)

let rec e (RHO rho) (PHI phi) =
 fun (* ... *)
  |  (CALL (f,l)) ->
     lookupphi f (map (e (RHO rho) (PHI phi)) l) phi

(* checkargs verify if two lists are equal *)

and checkargs = fun []       []    -> true
              |  (e1::l1) (e2::l2) -> (iseq e1 e2) && (checkargs l1 l2)
              |      _        _    -> false

(* member checks if an element is in a list or not *)

and member x = fun []   -> false
             |   (e::l) -> if (x=e) then true else member x l

(* lookupphi checks if one argument of the arg list is BOT. If it is the case,
   it returns BOT, otherwise it extracts the result value of f from phi *)

and lookupphi f arg phi = if (member (BOT) arg) then BOT
                          else lookupphi1 f arg phi

and lookupphi1 f arg = fun [] -> BOT
                     |  ((f1, arg1, res)::r) ->
                         if (checkargs arg arg1) && (f=f1)
                         then res
                         else lookupphi1 f arg r;;

(* Need: collects the function call that are needed *)

let rec need (RHO rho) (PHI phi) =
 fun NIL      -> []
 |   (INT i)  -> []
 |   (STR s)  -> []
 |   (VAR x)  -> []
```

```
 |   (FUNC (f,l)) ->
addneed ( f , (map (e (RHO rho) (PHI phi)) l) , (needlist (RHO rho) (PHI phi) l))
 |   (IF (c,a,b)) ->
      begin
      match (e (RHO rho) (PHI phi) c)
      with (I 0) -> (need (RHO rho) (PHI phi) c)@(need (RHO rho) (PHI phi) b)
      |    (I _) -> (need (RHO rho) (PHI phi) c)@(need (RHO rho) (PHI phi) a)
      |      _   -> (need (RHO rho) (PHI phi) c)
      end
 |   (SUCC x) -> (need (RHO rho) (PHI phi) x)
 |   (PRED x) -> (need (RHO rho) (PHI phi) x)
 |   (CAT (x,y)) -> (need (RHO rho) (PHI phi) y)@(need (RHO rho) (PHI phi) x)
 |   (EQU (x,y)) -> (need (RHO rho) (PHI phi) y)@(need (RHO rho) (PHI phi) x)
 |   (CONS(x,y)) -> (need (RHO rho) (PHI phi) y)@(need (RHO rho) (PHI phi) x)
 |   (CAR x) -> (need (RHO rho) (PHI phi) x)
 |   (CDR x) -> (need (RHO rho) (PHI phi) x)
 |   (ATOM x) -> (need (RHO rho) (PHI phi) x)

(* needlist computes need for a list of expressions *)

and needlist (RHO rho) (PHI phi) = fun []   -> []
                                 |   (e::l) -> (need (RHO rho) (PHI phi) e)
                                               @(needlist (RHO rho) (PHI phi) l)

(* For function call, if one of the arg is BOT then don't add the closure
   to C *)

and addneed (f, arg, nlist) = if (member BOT arg) then nlist
                              else (f,arg)::nlist;;

(* Some useful functions *)

(* Test the equality of two elements from phi *)

let iseq_phi (f1, l1, v1) (f2, l2, v2) =
              (f1 = f2) && (iseq v1 v2) && (checkargs l1 l2);;

(* Test if an element is in phi or not *)

let rec is_in_phi_list v = fun   []   -> false
                         |     (e::l) -> (iseq_phi v e)
                                         or (is_in_phi_list v l);;

(* Is a set included in another one ? *)

let rec included = fun [] [] -> true
                 | (e::l) m -> (is_in_phi_list e m) && (included l m)
                 |   []   _ -> true;;

(* Test if two phi sets are equal *)

let testphi l1 l2 = (included l1 l2) && (included l2 l1);;

(* Test the equality of two elements from C *)
```

```
let iseq_c (f1, l1) (f2, l2) = (f1 = f2) && (checkargs l1 l2);;

(* Test if an element from C is in C or not *)

let rec is_in_c_list v = fun [] -> false
                  |    (e::l)  -> (iseq_c v e) or (is_in_c_list v l);;

(* Is a set included in another one ? *)

let rec included_c = fun [] []   -> true
                  |    (e::l) m -> (is_in_c_list e m) && (included_c l m)
                  |     []  _   -> true;;

(* Test the equality of two C sets *)

let test_c l1 l2 = (included_c l1 l2) && (included_c l2 l1);;

(* Function that computes fix-point *)
(* phi and C are improved until they are equal *)

let rec iterate phi c prog =
    let phi1 = newphi c phi prog and
        c1   = union (newc c phi prog) c
    in if ((testphi phi phi1) && (test_c c c1)) then (phi1, c1)
       else iterate phi1 c1 prog

(* newphi computes a new phi *)
(* if C is the empty set, then returns the empty set
   otherwise, for each closure in C, we construct a phi *)

and newphi = fun    []      phi prog -> []
           |   ((f,arg)::r) phi prog ->
           (f, arg, e (RHO (buildrho f prog arg)) (PHI phi) (findexp f prog))::
           (newphi r phi prog)

(* newc computes a new C *)
(* if C is the empty set, then returns the empty set
   otherwise, for each closure in C, we collect new closures *)

and newc = fun [] phi prog -> []
         | ((f,arg)::r) phi prog ->
           (need (RHO (buildrho f prog arg)) (PHI phi) (findexp f prog))@
           (newc r phi prog)

(* buildrho constructs a rho for a function f in a program prog with a the list
   of value arg *)

and buildrho = fun f [] arg -> raise (EmptyList "in buildrho")
             |     f ((f1, lvar, exp)::r) arg ->
                     if (f=f1) then (zip lvar arg)
                     else buildrho f r arg

(* findexp returns the expr associated to a function f from a program prog *)
```

```
and findexp = fun f [] -> raise (EmptyList "in findexp")
            |     f ((f1, lvar, exp)::r) ->
                    if (f=f1) then exp
                    else findexp f r;;

(* start is a userfriendly way of calling iterate *)

let start f arg (PROG prg) = iterate [] [(f,arg)] prg ;;
```

B.6 A strictness interpreter using mfg

```
(* Compare this case with the interpreter using minimal function graph *)
(* New domain for a strictness interpreter
   We work with the two point domain 2 which contains ZERO and ONE *)

type bval = ZERO | ONE;;

(* Primitives for the strictness interpreter are only the logical 'and' and
   'or' *)

let myand = fun ONE ONE -> ONE
          |      _   _  -> ZERO;;

let myor = fun ZERO ZERO -> ZERO
         |      _    _   -> ONE;;

(* Some useful functions *)

let rec valofs s = fun []       -> raise (EmptyList "in valofs function")
                 |   ((a,b)::l) -> if (s=a) then b else valofs s l ;;

let rec zip = fun [] [] -> []
            |    (a::r) (b::s) -> (a,b)::(zip r s)
            |     _  _  -> raise (LengthError "in zip function");;

let rec iseq = fun (ONE) (ONE) -> true
             |     (ZERO) (ZERO) -> true
             |       _     _   -> false;;

(* E# *)

let rec esharp (RHO rho) (PHI phi) =
 fun NIL          -> ONE
  |  (INT i)      -> ONE
  |  (STR s)      -> ONE
  |  (VAR s)      -> (valofs s rho)
  |  (CALL (f,l)) ->
     lookupphi f (map (esharp (RHO rho) (PHI phi)) l) phi
  |  (IF (c, a, b)) ->  myand (esharp (RHO rho) (PHI phi) c)
                              (myor (esharp (RHO rho) (PHI phi) a)
                                    (esharp (RHO rho) (PHI phi) b))
  |  (SUCC x)     -> (esharp (RHO rho) (PHI phi) x)
```

```
  |  (PRED x)     -> (esharp (RHO rho) (PHI phi) x)
  |  (CAT (x,y))  ->
       myand (esharp (RHO rho) (PHI phi) x)
             (esharp (RHO rho) (PHI phi) y)
  |  (EQU (x,y))  ->
       myand (esharp (RHO rho) (PHI phi) x)
     (esharp (RHO rho) (PHI phi) y)
  |  (CONS (x,y)) ->
       myand  (esharp (RHO rho) (PHI phi) x)
      (esharp (RHO rho) (PHI phi) y)
  |  (CAR x) -> (esharp (RHO rho) (PHI phi) x)
  |  (CDR x) -> (esharp (RHO rho) (PHI phi) x)
  |  (ATOM x) -> (esharp (RHO rho) (PHI phi) x)

and checkargs = fun []       []    -> true
              |  (e1::l1) (e2::l2) -> (iseq e1 e2) && (checkargs l1 l2)
              |      _        _    -> false

and member x = fun []   -> false
             |   (e::l) -> if (x=e) then true else member x l

and lookupphi f arg phi = if (member (ZERO) arg) then ZERO
                          else lookupphi1 f arg phi

and lookupphi1 f arg = fun [] -> ZERO
                     |  ((f1, arg1, res)::r) ->
                        if (checkargs arg arg1) && (f=f1)
                        then res
                        else lookupphi1 f arg r;;

(* N# *)

let rec needsharp (RHO rho) (PHI phi) =
 fun NIL       -> []
 |   (INT i)   -> []
 |   (STR s)   -> []
 |   (VAR x)   -> []
 |   (CALL (f,l)) -> addneed ( f,
                               (map (esharp (RHO rho) (PHI phi)) l),
                               (needlist (RHO rho) (PHI phi) l) )
 |   (IF (c,a,b)) -> (needsharp (RHO rho) (PHI phi) c)@
                     (needsharp (RHO rho) (PHI phi) a)@
                     (needsharp (RHO rho) (PHI phi) b)
 |   (SUCC x)     -> (needsharp (RHO rho) (PHI phi) x)
 |   (PRED x)     -> (needsharp (RHO rho) (PHI phi) x)
 |   (CAT (x,y))  -> (needsharp (RHO rho) (PHI phi) y)@
                     (needsharp (RHO rho) (PHI phi) x)
 |   (EQU (x,y))  -> (needsharp (RHO rho) (PHI phi) y)@
                     (needsharp (RHO rho) (PHI phi) x)
 |   (CONS(x,y))  -> (needsharp (RHO rho) (PHI phi) y)@
                     (needsharp (RHO rho) (PHI phi) x)
 |   (CAR x)      -> (needsharp (RHO rho) (PHI phi) x)
 |   (CDR x)      -> (needsharp (RHO rho) (PHI phi) x)
 |   (ATOM x)     -> (needsharp (RHO rho) (PHI phi) x)
```

```
and needlist (RHO rho) (PHI phi) = fun []    -> []
                                 |   (e::l) -> (needsharp (RHO rho) (PHI phi) e)
                                        @(needlist (RHO rho) (PHI phi) l)

and addneed (f, arg, nlist) = if (member ZERO arg) then nlist
                              else (f,arg)::nlist;;

(* Fixpoint computation does not change with respect to the mfg
   implementation *)
```